Spring 2004

An approximate search engine for structure

Huiyuan Shan
New Jersey Institute of Technology

ABSTRACT

AN APPROXIMATE SEARCH ENGINE FOR STRUCTURE

by
Huiyuan Shan

As the size of structural databases grows, the need for efficiently searching these databases arises. Thanks to previous and ongoing research, searching by attribute-value and by text has become commonplace in these databases. However, searching by topological or physical structure, especially for large databases and especially for approximate matches, is still an art.

In this dissertation, efficient search techniques are presented for retrieving trees from a database that are similar to a given query tree. Rooted ordered labeled trees, rooted unordered labeled trees and free trees are considered. Ordered labeled trees are trees in which each node has a label and the left-to-right order among siblings matters. Unordered labeled trees are trees in which the parent-child relationship is significant, but the order among siblings is unimportant. Free trees (unrooted unordered trees) are acyclic graphs. These trees find many applications in bioinformatics, Web log analysis, phyloinformatics, XML processing, etc.

Two types of similarity measures are investigated: (i) counting the mismatching paths in the query tree and a data tree, and (ii) measuring the topological relationship between the trees. The proposed approaches include storing the paths of trees in a suffix array, employing hashing techniques to speed up retrieval, and counting the number of up-down operations to move a token from one node to another node in a tree. Various filters for accelerating a search, different strategies for parallelizing these search algorithms and applications of these algorithms to XML and phylogenetic data management are discussed.

The proposed techniques have been implemented into a phylogenetic search engine which is fully operational and is available on the World Wide Web. Experimental

results on comparing the similarity measures with existing tree metrics and on evaluating the efficiency of the search techniques demonstrate the effectiveness of the search engine. Future work includes extending the techniques to other structural data, as well as developing new filters and algorithms for speeding up searching and mining in complex structures.

AN APPROXIMATE SEARCH ENGINE FOR STRUCTURE

by
Huiyuan Shan

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Department of Computer Science

May 2004

APPROVAL PAGE

AN APPROXIMATE SEARCH ENGINE FOR STRUCTURE

Huiyuan Shan

Dr. Jason T. L. Wang, Dissertation Advisor — Date
Professor of Computer Science, NJIT

Dr. James A. McHugh, Committee Member — Date
Professor of Computer Science, NJIT

Dr. Chengjun Liu, Committee Member — Date
Assistant Professor of Computer Science, NJIT

Dr. Qun Ma, Committee Member — Date
Assistant Professor of Computer Science, NJIT

Dr. Bin Tian, Committee Member — Date
Assistant Professor of Biochemistry and Molecular Biology, UMDNJ

BIOGRAPHICAL SKETCH

Author: Huiyuan Shan

Degree: Doctor of Philosophy

Date: May 2004

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computerl Science,
  New Jersey Institute of Technology, Newark, NJ, 2004
- Master of Computer Science,
  Northeastern University, China, 1996
- Bachelor of Computer Science,
  Anshan University of Science and Technology, China, 1993

Major: Computer Science

Presentations and Publications:

Huiyuan Shan and Jason T. L. Wang, "Approximate Searching in Trees: Algorithms and Applications", submitted to *Knowledge and Information Systems.*

Jason T. L. Wang, Huiyuan Shan, Dennis Shasha and William H. Piel, "Fast Structural Search in Phylogenetic Databases ", submitted to *Applied Bioinformatics.*

Jason T. L. Wang, Huiyuan Shan, Dennis Shasha and William H. Piel, "TreeRank: A Similarity Measure for Nearest Neighbor Searching in Phylogenetic Databases", In *Proceedings of the 15th International Conference on Scientific and Statistical Database Management*, Cambridge, Massachusetts, July 2003, pp. 171-180.

Huiyuan Shan, Katherine G. Herbert, William H. Piel, Dennis Shasha and Jason T. L. Wang, "A Structure-Based Search Engine for Phylogenetic Databases," In *Proceedings of the 14th International Conference on Scientific and Statistical Database Management* , Edinburgh, Scotland, July 2002, pp. 7-10.

Dennis Shasha, Jason T. L. Wang, Huiyuan Shan and Kaizhong Zhang, "ATreeGrep: Approximate Searching in Unordered Trees," In *Proceedings of the 14th International Conference on Scientific and Statistical Database Management*, Edinburgh, Scotland, July 2002, pp. 89-98.

Katherine G. Herbert, Huiyuan Shan and Jason T.L Wang, "Approximate Searching in Phylogenetic Databases", In *Proceedings of the Atlantic Symposium on Computational Biology and Genome Information Systems and Technology*, Durham, North Carolina, March 2001, pp. 140-143.

This dissertation is dedicated to my beloved family

ACKNOWLEDGMENT

I am deeply grateful to my advisor, Dr. Jason T. L. Wang for his constant guidance to my research, advice and encouragement that helped me through the way of Ph.D. study, and the time spent on reviewing my papers and dissertation drafts. I would like to thank Dr. William Piel, the designer of TreeBASE, for his valuable feedback and collaboration.

I want to express my appreciation to Dr. James A. McHugh, Dr. Chengjun Liu, Dr. Qun Ma, Dr. Bin Tian, Dr. William H. Piel and Dr. Wynne Hsu for serving as members of the dissertation committee.

I am grateful for the financial support I received through my studies. This work was supported in part by U.S. NSF grants IIS-9988345 and IIS-9988636.

Finally special thanks go to my family, especially my wonderful wife for her understanding and encouragement.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Chapter **Page**

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

LIST OF FIGURES
(Continued)

Figure **Page**

LIST OF FIGURES
(Continued)

Figure **Page**

CHAPTER 1

INTRODUCTION

Labeled trees represent data in many scientific and commercial disciplines. For example, scientists model phylogenetic relations as unordered labeled trees and develop methods for constructing these trees [2, 9, 33]. A recent workshop report from Yale suggested that more research be undertaken to improve heuristic search strategies using algorithms designed to meet the demands made by increasingly large tree datasets [44]. Recent efforts in Web computing model an XML document as a ordered tree, offering further motivation for the need for efficient tree searching [3, 10]. Free tree searching is of considerable interest in many computer version problems [49, 40]. The molecule is also described by a free tree structure representing its major chemical building blocks and the way they are connected [51]. The free tree representation of molecule is valuable for searching new drugs in the molecule database. Other potential applications include information retrieval in linguistic, taxonomic, and neuroanatomical databases, among others.

Many algorithms have been developed for tree searching and matching [1, 26, 43, 59]. Most of these algorithms focus on comparing two trees based on various distance metrics. Chawathe *et al.* [11, 12, 13] studied the tree matching problem in the context of change detection for structured and semistructured data. There are also efforts spent in the development of query languages [31, 39, 56, 57] and query processing techniques [14] for trees, with applications to XML and object-oriented database management.

In the study of phylogeny, the phylogenetic tree is essential for understanding the relationship between the organisms, or taxa, involved within the study. Many of the current techniques for searching phylogenetic repositories allow the user to search

using one of three ways: through a keyword-type search; through sequence alignment; or through browsing a hierarchical list of taxa. The new search engine described in this dissertation allows the user to present an example, or a query, tree and then searches a phylogenetic database for trees that (approximately) contain the query structure. The presented search engine is fully operational and is available on the World Wide Web.

The rest of the dissertation is organized as follows. Chapter 2 discusses some background knowledge for structural search on phylogenetic trees. Chapter 3 discusses our proposed path based method. Chapter 4 describes our new similarity measure for NN search on TreeBASE. Chapter 5 introduces the web-based system and prototypes. Chapter 6 discusses future work and concludes the dissertation.

CHAPTER 2

BACKGROUND

2.1 Phylogenetic Tree

In this chapter, some background knowledge for structure search on phylogenetic trees is discussed.

The phylogenetic tree models the evolutionary history of a set of taxa that have a common ancestor. It is usually represented by a dendrogram. The internal nodes within a particular tree represent older organisms from which their child nodes descended. The children represent divergences in the genetic composition in the parent organism. Since these divergences cause new organisms to evolve, these organisms are shown as children of the previous organism.

Currently, in studying the phylogenetic data, most systems use search methods that do not exploit the structure of the data. These systems usually adopt a keyword-based search tool, a sequence alignment, or a manually operated browser. The keyword-based tool allows the user to enter the name of a taxon or some identifying quality such as an identification number to search the database. It then returns all the data it has stored on that particular taxon. Some systems even allow the user to search more than one taxon at a time, using the traditional "AND" and "OR" operators to decide what set of data to return to the user. In sequence alignment, the taxon uses the genetic code from the sequence to align it with similar sequences. If the sequences are similar, most likely they have a common ancestor. A phylogenetic tree can then be obtained from that clue. Once the taxa are selected that align with the query taxon, links to other information about the returned taxa can be provided. Finally, browsing a hierarchical list allows users to examine one taxon at a time, learning only about that taxon.

Many of the systems that employ the search methods described above include visualization techniques that allow the user to view an entire section of a phylogenetic tree or the entire tree as well as interact with it. These interactions can include visually inspecting the tree for relationships between taxa as well as linking to other trees that contain a particular taxon or viewing isomorphism trees so that relationships between the taxa can be better viewed.

However, none of the existing systems provides the user with the ability to search a database for the structure of a phylogenetic tree or structures similar to a query structure. Since the structure of a phylogenetic tree models very important information about the taxa contained within the tree, structure search becomes a very helpful as well as important tool for researchers studying phylogeny. While visual inspection of the structure of a tree can help researches to learn a great amount of information about the relationship between taxa, if the tree is large, visual identification can become unwieldy at best and even impossible if the tree is large enough. As mentioned, the phylogenetic tree models the relationships between taxa. By being able to query based on a tree's structure, a user can query by the relationship among taxa. This is extremely effective for a researcher who desires to learn more about multiple taxa and where and how their evolutionary development diverges.

2.2 TreeBASE

TreeBASE, accessible at http://www.herbaria.harvard.edu/treebase, is a relational database containing phylogenetic information from research papers submitted to the Web site. The site then allows users to search the database freely according to various keywords, and see visual representations of the trees. Moreover, it allows the user to gain access to information concerning a tree as well as use comparison tools to learn more about various taxa contained within the tree and their relationships with other taxa within the database.

The dataset TreeBASE maintains consists of phylogenetic trees submitted to TreeBASE by the authors of the papers that present the trees. The site accepts for review any peer-reviewed and published paper that presents information on any type of phylogenetic trees. For the paper to be contained within the database, it must be submitted to the site. The paper then goes through a review process before it is officially put within the database.

The schema and relational tables in TreeBASE contain various types of data including the citations of the papers stored within the database, the abstracts from the papers, the information about the authors, the algorithm used to obtain the phylogenetic trees, the titles and types of the trees, the software used to perform the relevant analysis, the association of the trees and matrices with the study through which they were obtained, and the information about the taxa.

TreeBASE allows the user to search its database by various keywords, including taxa, author, citation, study accession and matrix accession. Search results contain the information about the study that an input keyword was found in. This information includes the publishing date, the author, the title of the study in which the keyword was found, and the periodical in which the study appeared. Also, accompanying the study are analyses of the data presented within the study. These analyses can include the matrix from which the phylogenetic trees are generated, a link to drawing the tree in a frame within the Web site, a link to download the tree so that the user can view it on his or her own viewer, and a link to "mark" a tree which allows the user to store the tree for quick retrieval later. TreeBASE is also equipped with various visualizations tools for drawing and displaying trees, and allows the user to "tree surf" and download the matrix of a particular tree.

CHAPTER 3

ATREEGREP: APPROXIMATE SEARCHING IN TREES

3.1 Introduction

A new approach is proposed for approximate search among labeled trees. This problem, denoted the *approximate nearest neighbor search* (ANN) problem for labeled trees, is the following. Given an integer $DIFF$, a query tree Q and a database $\mathcal{D}$ of trees, the ANN problem is to find all the data trees D in $\mathcal{D}$ where D approximately contains Q within distance $DIFF$. That is, D contains a substructure D' and the distance from Q to D' is at most $DIFF$. It is proved that this problem was NP complete [68] for edit distance between unordered trees. So a different method is adopt. The distance from Q to D' is measured by the total number of root-to-leaf paths in Q that do not appear in D'; the nodes in D' that do not appear in Q can be freely removed. No previous work has addressed the ANN problem for labeled trees.

To illustrate the distance measure, consider the rooted query tree Q in Figure 3.1, which has two paths. Unordered rooted trees are trees in which the parent-child relationship is significant, the order among siblings is unimportant. For the ordered rooted trees, the distance is subject to the order constraint. For the unordered cases, the query tree will match data tree D_1 with distance of 0, and match data tree D_2 with distance of 1. This happens because every path in Q matches a path of the substructure D_1' in D_1. (D_1' is enclosed by the dashed line in D_1.) On the other hand, there is one path '$a - e$' in Q that cannot be found in D_2. Counting mismatching paths is important in, for example, inferring evolutionary history, since that shows a deviation of ancestor-descendant relations. It is also a natural extension of path expressions in XML queries.

This distance metric is used for two reasons, one semantic and one pragmatic:

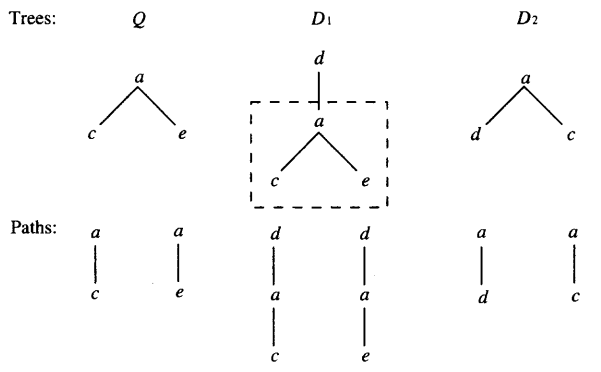


Figure 3.1 Example rooted trees.

1. In trees, the parent-child relationship is the most significant one. This is reflected in paths. For example, each path in a phylogenetic tree stands for the evolutionary history of a taxon. When several siblings may have the same label, post-processing must determine whether two paths of the form a – b – c and a' – b – c pertain to the same b or different bs.

2. The pragmatic reason is that there exist efficient algorithms for string searching. By decomposing trees to paths and by transforming tree searching to string searching, one can take advantage of the existing string searching algorithms and perform structural search efficiently.

In practice, it's likely that some portion of a query tree is unknown, uninteresting or unimportant. That portion is often represented by a don't care symbol. In general, there are two types of don't care symbols: variable length don't cares (VLDCs) [62, 67] and fixed length don't cares (FLDCs). In string matching, a VLDC, denoted "*", in the query string may substitute for zero or more characters in a data string. For example, if "com*er" is the query string, then the "*" would substitute for the substring "put" when matching with the data string "computer". On the other hand, a FLDC, denoted "?", in the query string substitutes for exactly a single character in a data string. For example, if "com?uter" is the query string, then the "?" would substitute for the character "p" in the match with the data string "computer".

In this chapter generalizations of don't cares to trees and algorithms for processing them are discussed. In these cases, the labels on nodes can be "$*$" or "?".

3.2 Basic Algorithms

The new pathfix algorithm, consists of two phases. In the first phase, a suffix array database $\mathcal{SD}$ is built for all the trees in the database $\mathcal{D}$. $\mathcal{D}$ contains strings where each string corresponds to a root-to-leaf path in a data tree. The paths (strings) are encoded into a suffix array [41]. In the second phase, which is the on-line search phase, The root-to-leaf paths of the query tree Q are compared with the paths in the suffix array database $\mathcal{SD}$ to locate those substructures approximately matching the query tree. In a later enhancement, a filter is constructed to determine which data trees in $\mathcal{D}$ are possible matches.

3.2.1 Indexing Trees

The suffix array is a data structure designed for efficient searching in a large string [41]. This data structure is simply an array containing the pointers to all the string's suffixes sorted in lexicographical order. (A suffix is a substring starting at a certain position in the string and ending at the end of the string.) Searching for a query string can be performed by binary search using the suffix array.

In pathfix, a suffix array is constructed for all the paths in a data tree and put it in a global set of suffix arrays for all the data trees. This global set is the database $\mathcal{SD}$. Figure 3.2 shows the algorithm.

As an example, consider again the data tree D_1 in Figure 3.1. D_1 has paths '$d-a-c$' and '$d-a-e$'. A suffix array SA_1 is created for the two paths, separated by a delimiter #, as shown in Figure 3.3. In this figure, the parenthesized integer in front of each suffix indicates the position at which the suffix begins in the paths set. This integer, when stored in the suffix array, serves as a pointer to the corresponding

Procedure Build_Suffix_Array
Input: the database $\mathcal{D}$ of trees.
Output: the suffix array database $\mathcal{SD}$.

1. **for** every tree D in the database $\mathcal{D}$
2. find all the paths in D;
3. concatenate those paths with a delimiter to form a long string S;
4. form a suffix array for S and add it to the global set of suffix arrays;
5. **end for**;
6. return the global set of suffix arrays, which is the database $\mathcal{SD}$;

Figure 3.2 Procedure for building the suffix array database $\mathcal{SD}$.

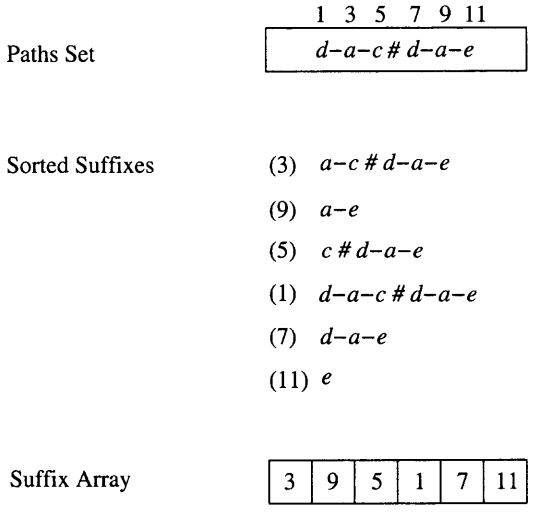


Figure 3.3 A suffix array.

suffix in the paths set. Likewise a suffix array SA_2 can be created for the paths of the data tree D_2 in Figure 3.1. The suffix array database $\mathcal{SD}$ contains SA_1 and SA_2.

For rooted trees, the paths of the rooted tree is all the paths from root to leaves. In the unrooted case, the path of the unrooted tree is all the paths between any two terminal nodes. The node with degree of 1 is a terminal node. If the path from terminal node a to terminal node b is reversed, the path from terminal node b to terminal node a can be got , and vice versa. So in the database of unrooted trees, the path between a and b is kept only once.

Lemma 3.1 *Suppose each data tree has at most N nodes, and there are M trees in the database $\mathcal{D}$. The space complexity of the procedure* Build_Suffix_Array *in Figure 3.2 is $O(MN^2)$. The time complexity is also $O(MN^2)$.*

Proof. Each rooted data tree has at most $O(N)$ paths if the tree is very bushy. No path can be longer than $O(N)$. At most N^2 pointers are needed to the suffixes of the paths. The worst case for free tree is also $O(N^2)$. Thus each data tree requires $O(N^2)$ space. There are totally M trees, so the space complexity is $O(MN^2)$. The expected time spent in constructing a suffix array for a string is linearly proportional to the string size [41], so the database $\mathcal{SD}$ can be built in $O(MN^2)$ expected time.

In practice, a tree with N nodes either has few paths or short depth, so the above upper bound is very pessimistic. In practice the complexity is linear, i.e. $O(MN)$.

There are two alternative implementations of the suffix arrays. One suffix array can be constructed for all the trees or one suffix array can be constructed for each tree (as in the current implementation). Choosing one or the other depends on the effectiveness of filtering. If filtering yields very few candidate trees, then the current implementation works well. If there are many similar trees in the database, then a single suffix array for all trees is better.

3.2.2 Search on Rooted Unordered Trees

In the on-line search phase, the query tree Q is compared to each data tree allowing a difference $DIFF$. When comparing Q with a data tree D, every root-to-leaf path p in Q is taken and roots of that path in D are found. (As a cutoff optimization, searching D is stop if more than $DIFF$ paths of Q are not found in D.) Suppose there are k root-to-leaf paths in Q. If a node n in D is the root of the k paths, then the subtree D' rooted at n matches Q with distance 0, provided there are no siblings having the same label. (If there are siblings having the same label, then post-processing can verify the match. This technique will never miss a match.) If n

is the root of $k-1$ paths, then D' matches Q with distance 1 and D approximately contains Q with distance 1.

Figure 3.4 shows the algorithm. Note that this algorithm can easily be modified to print out the subtree D' rooted at n that matches Q.

Procedure Basic_Unordered_Search
Input: the allowed distance threshold $DIFF$, the query tree Q, the database $\mathcal{D}$ of unordered trees and the suffix array database $\mathcal{SD}$.
Output: the set $\mathcal{R}$ of data trees that approximately contain Q within distance $DIFF$.

1. $\mathcal{R} := \emptyset$;
2. compute all the root-to-leaf paths of the query tree Q;
3. let k be the number of paths of Q;
4. **for** each data tree D in $\mathcal{D}$ (after filtering);
   /* Suffix array search portion */
5. **for** each path p in Q from longest to shortest
6. find the root set N_p in D such that for each n in N_p there is a node n' and the path from n' to n (ascending in D) is p;
7. exit the for loop if the root sets for more than $DIFF$ paths are empty;
8. **end for**;
   /* Intersection portion */
9. **for** each n in N_p
10. count the total number of occurrences $T(n)$ of n in all root sets N_p's for all paths p in Q;
11. **if** $T(n) \geq k - DIFF$ **then**
12. $\mathcal{R} := \mathcal{R} \cup \{D\}$;
13. **end for**;
14. **end for**;

Figure 3.4 Procedure for finding unordered data trees approximately containing the query tree Q.

Lemma 3.2 *Let q be the number of nodes in the query tree Q. Suppose that the size of a suffix array is S. The time complexity of the suffix array search portion of procedure* Basic_Unordered_Search *in Figure 3.4 for such a suffix array is $O(q^2 \log S)$.*

Proof. q is an upper bound on the number of paths in Q; q is also an upper bound on the lengths of the paths in Q. Searching for a path of length q takes time $O(q \log S)$, and hence the result follows.

The time spent to count the number of occurrences of the entire tree depends on the number of matches of the paths. In the worst case, this can be nearly every node, but in practice is much smaller.

3.2.3 Search on Rooted Ordered Trees

Ordered tree makes distinction among the various children of a node, such as "first child" or "last child". $v_1 < v_2$ if v_1 is the left sibling of v_2. Path precedence ($<$) is defined as $P1 < P2$ if there exist $v1$ on $P1$ and $v2$ on $P2$ such that $v1 < v2$. Path sequence, S(Tx) is the path precedence sequence of all paths starting from x, where node x is in T. There exists a optimal alignment of two path sequence of query tree and S(Tx). The distance between Q and D is the minimum number of paths in Q that do not align with S(Tx), where x is in D. For the ordered cases in Figure 3.1, the query tree will match data tree both D_1 and D_2 with distance of 1.

The ordered tree search is base on unorder tree search. First the data trees are filtered by searching ordered trees as unordered trees, then the path order among those candidate trees is checked. A sequence alignment method is adapted to check the path order.

To illustrate the search on ordered trees, a precedence relation among paths starting from the same root needs to be defined. For any two paths $P1$ and $P2$ that start from the same node c, there must be two siblings along the two paths, let them be a in $P1$ and b in $P2$. If a precedes b, then $P1$ precedes $P2$, otherwise $P1$ succeeds $P2$.

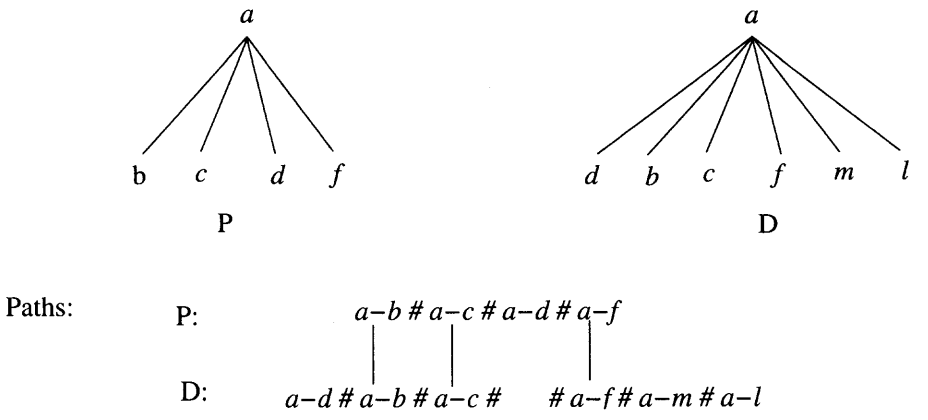


Figure 3.5 Illustration of matching a query tree with a ordered data tree within distance of 1.

Lemma 3.3 *Let D be one of the search result trees for the query tree Q within distance threshold $DIFF$. D must satisfy: unordered_dist(Q, D) <= ordered_dist(Q, D)*

Proof. Ordered trees is a special kind of unordered trees. If a data tree satisfies the threshold $DIFF$ in an ordered way, it must satisfies the threshold $DIFF$ in an unordered way. If ordered trees are searched as unordered trees by procedure Basic_Unordered_Search, all data trees satisfying the threshold $DIFF$ in an unordered way are returned, so additional ordered checking is needed to verify those candidate trees.

After the paths are concatenated according to their precedences, the procedure of order checking among those paths is the same as finding maximum alignment among two sequences. Hence the maximum sequence alignment algorithm [28] is adapted to verify those candidate trees. Figure 3.5 shows an example. A detailed algorithm is shown in Figure 3.6

Lemma 3.4 *Let q be the number of nodes in the query tree Q. n is the upper bound of number of paths in the data tree that match the paths of the query tree. Suppose that the size of a suffix array is S. The time complexity of the suffix array*

Procedure Basic_Ordered_Search
Input: the allowed distance threshold $DIFF$, the query tree Q, the database $\mathcal{D}$ of ordered trees and the suffix array database $\mathcal{SD}$.
Output: the set $\mathcal{R}$ of data trees that approximately contain Q within distance $DIFF$.

```
1.  R := ∅;
2.  compute all the root-to-leaf paths of the query tree Q;
3.  let k be the number of paths of Q;
4.  for each data tree D in D (after filtering);
        /* Suffix array search portion */
5.      for each path p in Q from longest to shortest
6.          find the root set N_p in D such that for each n
            in N_p there is a node n' and the path from n'
            to n (ascending in D) is p;
7.          exit the for loop if the root sets for more than
            DIFF paths are empty;
8.      end for;
        /* Intersection portion */
9.      for each n in N_p
10.         count the total number of occurrences T(n)
            of n in all root sets N_p's for all paths p in Q;
11.         if T(n) ≥ k − DIFF then
12.             Let m be the number of matched paths in query tree
                and n be the matched paths that start from root t
13.             Produce a matched path matrix Match [m,n]
14.             for (i=1;i≤m,i++)
15.                 for (j=1;j≤n, j++)
16.                     MaxPath[i,j]=MaxPath[<i,<j]+ Match[i,j];
17.                     if MaxPath[m,n] ≥ k − DIFF then
18.                     R := R ∪ {D};
19.     end for;
20. end for;
```

Figure 3.6 Procedure for finding ordered data trees approximately containing the query tree Q.

search portion of procedure Basic_Ordered_Search *in Figure 3.6 for such a suffix array is* $O(q^2 \log S)$ *and the order checking portion is* $O(qn)$.

Proof. From lemma 3.2, finding candidate trees as unordered trees is $O(q^2 \log S)$. The order checking is $O(mn)$, and q is also an upper bound of m, so the time for order checking is $O(qn)$. The order checking takes a small portion of the total running time. In practice,Basic_Ordered_Search runs as fast as Basic_Unordered_Search.

3.2.4 Extensions to Free Trees

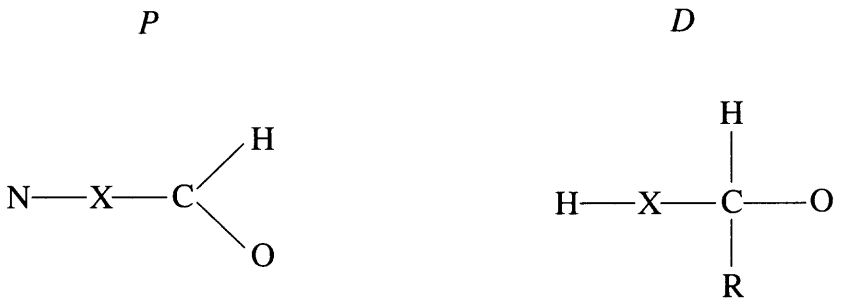
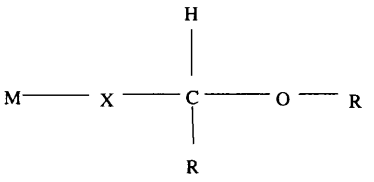


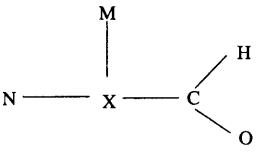
Figure 3.7 Example unrooted trees.

In free trees, the path from a node x to any terminal node is considered to be path of x. A configuration of a free tree T is the rooted unordered tree of T_n where root is n (n is in T). For unrooted free trees, the distance from query tree to data tree is defined to be Min (dist(Q_n, G_n')), where Q_n is a configuration of Q and $G_{n'}$ is a configuration of G. For example in Figure 3.7, The distance from P to D is 1. For the three paths starting from "X", two of them can find match in D, and the path X-N can not be found in D.

In the database, only the path between one terminal node and another terminal node is encoded in the suffix array. In addition, the path is stored only once. In another word, the paths of unrooted trees in the database have no direction. In the search phrase, the Free tree search algorithm scans the path in the database in two direction. The goal of search on free trees is to minimize the total number of



D



P

Paths in D:
D1: H–C–X–M
D2: H–C–R
D3: H–C–O–R
D4: R–O–C–X–M
D5: R–O–C–R
D6: R–C–X–M

Paths in P orginated from X:
P1: X–M
P2: X–C–H
P3: X–C–O
P4: X–N

Figure 3.8 Illustration of matching a query tree with a free data tree within distance of 1.

mismatched paths and maximize the total length of all matched paths. A detailed algorithm is shown in Figure 3.9 In practice, the searching results are ranked according to their total length of their all matched paths. For example in Figure 3.8, by scanning the paths of data tree D from two direction, paths P1, P2 and P3 can find a match, while P4 can not find a match so the total distance from P to D is 1.

Lemma 3.5 *Let q be the number of nodes in the query tree Q. n is the upper bound of number of paths in the data tree that match the paths of the query tree. Suppose that the size of a suffix array is S. The time complexity of the suffix array search portion of procedure* Basic_Unrooted_Search *in Figure 3.9 for such a suffix array is $O(q^3 \log S)$ and the order checking portion is $O(qn)$.*

Proof. From lemma 3.3, the procedure Basic_Unordered_Search takes $O(q^2 \log S)$. The procedure Basic_Unrooted_Search checked all nodes in the query tree, so the total running time is $O(q^3 \log S)$.

Procedure Basic_Unrooted_Search
Output: the set $\mathcal{R}$ of data trees that approximately contain Q within distance $DIFF$.

```
1.  for each node x in Q
2.  compute all the node-to-terminal paths of x;
3.  R := ∅;
4.  let k be the number of paths of x;
5.  for each data tree D in 𝒟 (after filtering);
        /* Suffix array search portion */
6.      for each path p of x from longest to shortest
7.          p' := reverse the path p;
8.          find the starting node set N_p in D such that for each n
            in N_p there is a node n' and the path from n'
            to n (ascending in D) is p or p';
9.          exit the for loop if the starting node sets for more than
            DIFF paths are empty;
10.     end for;
        /* Intersection portion */
11.     for each n in N_p
12.         count the total number of occurrences T(n)
            of n in all starting node sets N_p's for all paths p or p' in Q;
13.         if T(n) ≥ k − DIFF then
14.             R := R ∪ {D};
15.     end for;
16. end for;
```

Figure 3.9 Procedure for finding unrooted data trees approximately containing the query tree Q.

3.3 Advanced Algorithms

3.3.1 Filtering Trees

The search process can be heuristically improved by using hashing technique that works as follows on the non-wildcard portion of data and query trees. compute and store all individual node labels and all parent-child label pairs in each data tree into a hash table, associating each parent-child pair with the set of data trees that contain the parent-child pair. Now suppose a query tree Q is given with a certain distance allowed in searching, $DIFF$. Take the multiset of labels from Q and see which data trees have a super-multiset of those possibly with $DIFF$ missing labels. Take the multiset of parent-child pairs from Q and see which data trees have a super-multiset of those parent-child pairs, again with $DIFF$ missing pairs. This heuristic, referred to as pathfilter, eliminates irrelevant trees from consideration in the beginning of a search and yield a set of candidate trees to look for. For example in Figure 3.10, $DIFF$ is set to 0. Both T_1 and T_2 miss ac pair . so they will not be a solution and hence can be eliminated from consideration.

3.3.2 Query Trees with Don't Cares

When generalizing don't cares to trees, the semantics of the don't cares is as follows. The VLDC "$*$" in the query tree may substitute for a path of length zero or more in a data tree. The FLDC "?" in the query tree may substitute for a single node in the data tree. Figure 3.11 shows the algorithm for finding data trees exactly containing the query tree Q where the root of Q is not a don't care.

In general, for a query tree Q with don't cares, a node x in a data tree D is the root of a subtree that matches Q if all of the following hold:

1. The partition of Q containing the root r_{all} of Q (call that the root partition of Q) matches D at x.

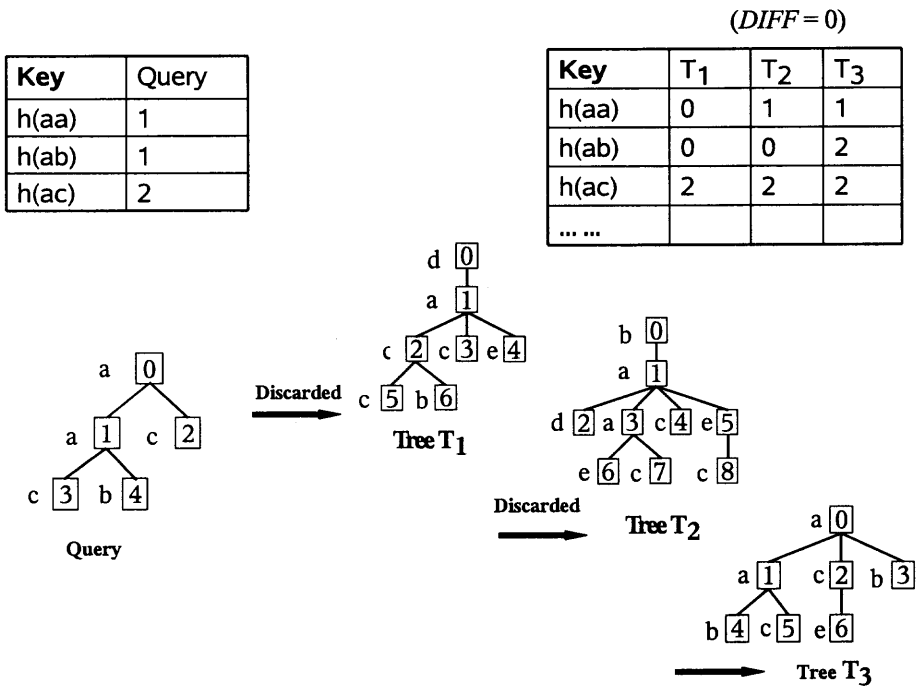


Figure 3.10 Illustration of matching a query tree having don't cares with a unordered data tree.

2. Consider the path p from the root r_{sub} of a subtree in Q to r_{all}. Suppose that r_{sub} matches D at possibly many nodes $x_1, x_2, \ldots$. The path from at least one such node in D, say x_j, has the property that the ascending path from x_j to x matches (with "$*$" and "?") the path from r_{sub} to r_{all}.

To avoid testing the roots of subtrees unnecessarily, the matching makes use of facts like the following: if Q is to match the data tree D at x, then the only relevant matches of a subtree of Q rooted at r_{sub} are nodes that are descendants of x.

Don't cares add to the time because each partition is much more likely to match than the whole tree, so there are many possible combinations to test. The basic time used for checking the suffix arrays, however, is less, since each tree is broken up into smaller trees.

Procedure Advanced_Search
Input: the query tree Q with don't cares, the database $\mathcal{D}$ of trees and the suffix array database $\mathcal{SD}$.
Output: the set of data trees containing Q and for each such data tree D, the substructure D' in D that matches Q.

1. partition Q into connected subtrees having no don't cares;
2. match each of those subtrees with data trees in $\mathcal{D}$ by invoking procedure Basic_Unordered_Search for Unordered data trees, Basic_Ordered_Search for Ordered data trees and Basic_Unrooted_Search for unrooted data trees respectively;
3. For the matched substructures that belong to the same data tree, say D, determine whether they combine, forming D', to match Q based on the matching semantics of the don't cares explained in the text, and if so, return D and D';

Figure 3.11 Procedure for finding data trees containing the query tree Q with don't cares.

Figure 3.12 illustrates how to match a query tree Q having don't cares with a data tree D. Q is first partitioned into three subtrees at its don't cares "$*$" and "?". The subtrees of Q match three subtrees in D, which are then glued. In the figure, nodes in D that are not touched by a dashed line are to be removed at no cost. The don't care "?" is instantiated into node o in D and the don't care "$*$" is instantiated into nodes h, j in D at no cost. The distance between Q and D is 0.

When a distance $DIFF$ is allowed in matching a query tree Q with a data tree, for each don't care free subtree Q' of Q, by invoking procedure Basic_Unordered_Search for Unordered data trees and Basic_Ordered_Search for Ordered data trees respectively, all subtrees of data trees are found, that are within distance $DIFF$ of Q'. The gluing process involves a testing of whether the glued tree as a whole is indeed within distance $DIFF$ of the entire query tree Q.

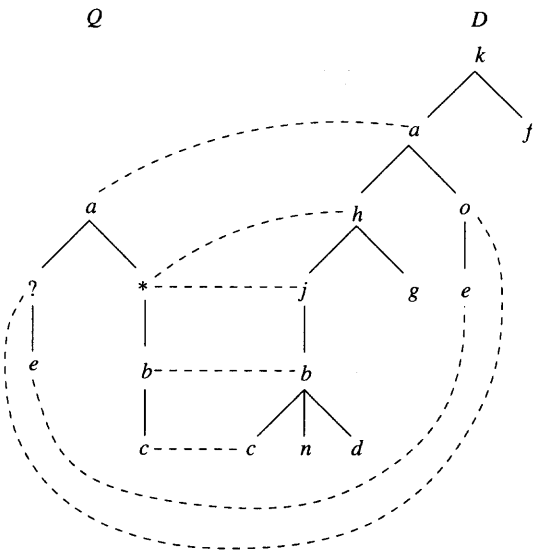


Figure 3.12 Illustration of matching a query tree having don't cares with a unordered data tree.

3.4 Experiments and Results

3.4.1 Sequential Implementation

A series of experiments have been conducted to evaluate the performance of the algorithms pathfix and pathfilter, collectively referred to as ATreeGrep (reminiscent of AGrep [66] for approximate string searching and SGrep [30] for structure grep). In this section, discussion is first focused on the performance of ATreeGrep one processor. The programs were written in C and run under Solaris on a Sun Blade 1000 workstation.

One thousand trees were randomly generated, each tree having 100 nodes. The number of branches on internal nodes ranges from 1 to 8 and the average length of paths in each trees ranges from 5 to 9. The string labels of nodes were randomly chosen from a dictionary. In each run, a tree was selected and modified into the query tree (with or without don't cares) and the other trees were used as data trees. Ten runs were tested and the average was plotted. Figure 3.13 and Figure 3.14 show the times spent in running ATreeGrep on the synthetic trees where the dictionary size is 50 and 1000, respectively.

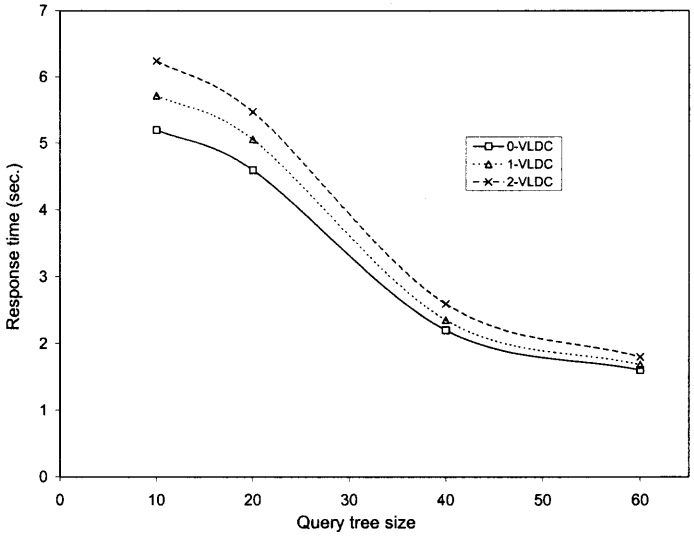


Figure 3.13 Running times of ATreeGrep on the 1000 synthetic trees with a label dictionary size of 50.

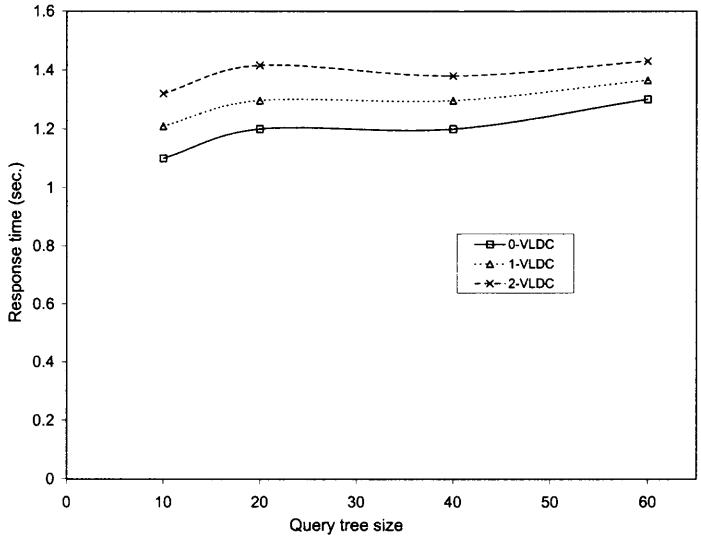


Figure 3.14 Running times of ATreeGrep on the 1000 synthetic trees with a label dictionary size of 1000.

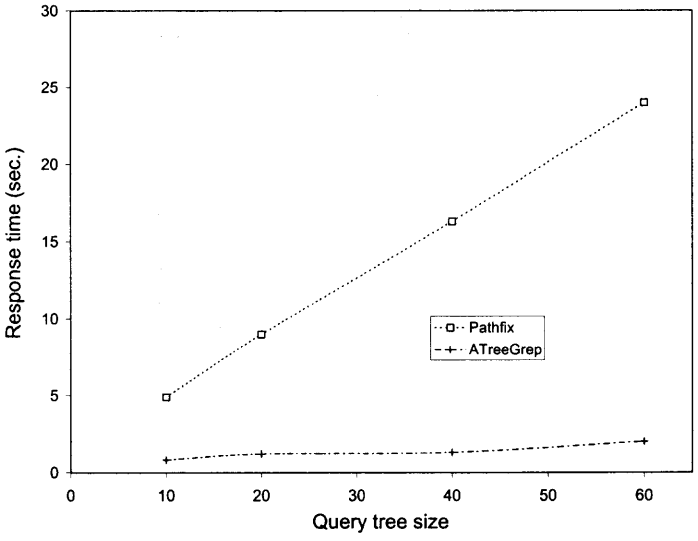


Figure 3.15 Running times of ATreeGrep and Pathfix on the 1000 synthetic trees with a label dictionary size of 1000.

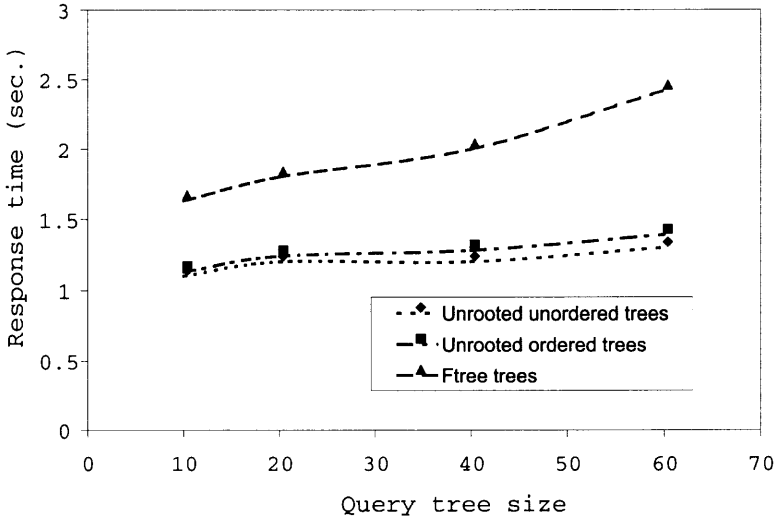


Figure 3.16 Running times of ATreeGrep on rooted unordered trees, rooted ordered trees and free trees on the 1000 synthetic trees with a label dictionary size of 1000.

It can be seen from Figure 3.13 and 3.14 that the dictionary size has a significant impact on the running times of ATreeGrep. When both the dictionary size and query tree are small, pathfilter finds many candidate trees, requiring much time for checking. When the query tree is large, however, the parent-child pairs combined with their counts produce a selective filter and hence there are few trees to look for. Consequently, the total running time decreases.

On the other hand, when the dictionary size is large, very few parent-child pairs are the same regardless of the query tree size. As a consequence, many trees are eliminated by pathfilter.

Figure 3.15 compares ATreeGrep and pathfix for varying query tree sizes, where the dictionary size was fixed at 1000. The figure shows that pathfilter speeds up ATreeGrep considerably. It can be seen that the running time of pathfix is proportional to the size of a query tree (actually the total number of paths of the query tree). It was also observed that the running time of pathfix is proportional to the number of matches in the database.

Figure 3.16 compares ATreeGrep on rooted unordered trees, rooted ordered trees and free trees for varying query tree sizes, where the dictionary size was fixed at 1000. It can be seen that the running time on rooted ordered trees is slightly more than that on rooted unordered trees. While search on free trees requires to check many configurations of the data trees, it take much more time to get the results.

The algorithms is tested on the phylogenetic trees obtained from TreeBASE maintained at Harvard University Herbaria (`http://www.herbaria.harvard.edu/treebase`). Phylogenetic trees are structures used in biology to study the evolution of various life forms as well as the relationship of a particular life form with other life forms. 1548 phylogenetic trees in TreeBASE are considered. Most non-leaf nodes in these trees have two children. The number of nodes of a tree ranges from 50 to 200, and the dictionary size of node labels is 18870. All the leaf-nodes and some non-leaf

nodes have labels. For each non-leaf node without a label, "U" is used as its label. Figure 3.17 shows the results. The results are promising and consistent with those for the synthetic data.

3.4.2 Parallel Implementation

The Pathfilter is used to speed up the sequential version on one computer. To scale beyond the capabilities of the serial implementation, the parallel approaches are exploited. With the developments in parallel and distributed computing, a number of parallel approaches have been developed to improve the performance of search algorithms[36, 22]. In the following discussion of parallelizing ATreeGrep, two approaches are focused on because of the specific characteristics of the tree search algorithms.

The main idea behind Partition Based Parallel ATreeGrep (PB-ATreeGrep) is to divide the search space among processors. First the data trees are partitioned into n partitions among n processors evenly. Each processor runs the serial ATreeGrep on it local partition, and broadcasts its results to a master processor, which collects the evaluate the result trees.

Although PB-ATreeGrep improves the performance over the serial ATreeGrep in almost all cases, because each processor runs both pathfilter and pathfix, some processors may filter out all trees in its space, and find out that it need not bother to run pathfix on its partition. It is highly possible that one processor finish searching its space far ahead another processor, so LB-ATreeGrep is designed and implemented to reduce the idle time.

The filter process takes only a small portion of the whole search. The pathfix is the most time consuming part in ATreeGrep. Load Balancing Parallel ATreeGrep(LB-ATreeGrep) offers a balanced division of work between all processors to reduce idle time and minimize the redundant work. Each processor stores the entire database in

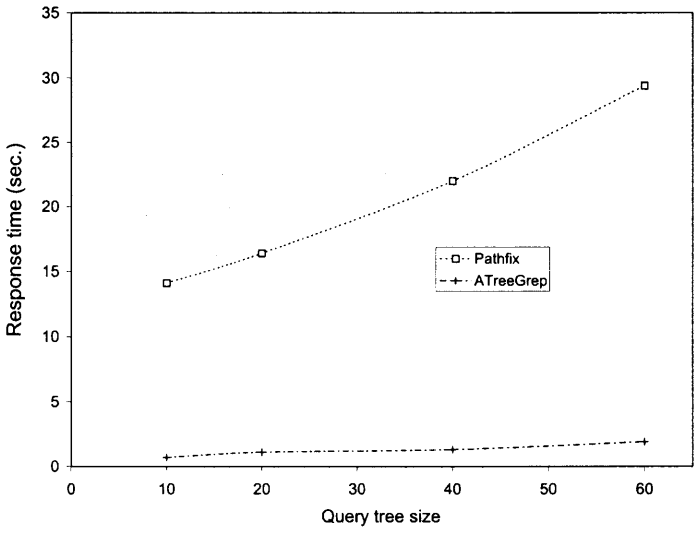


Figure 3.17 Running times of ATreeGrep and Pathfix on the 1548 phylogenetic trees obtained from TreeBASE.

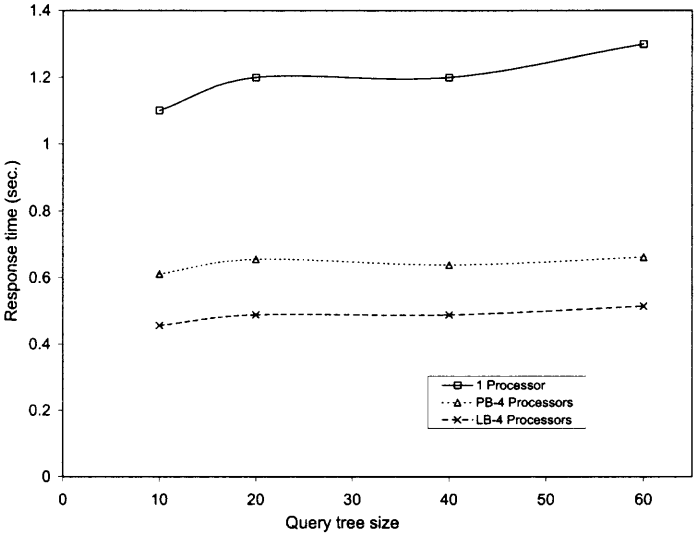


Figure 3.18 Running times of serial ATreeGrep , PB-ATreeGrep and LB-ATreeGrep on the 1000 synthetic trees with a label dictionary size of 1000.

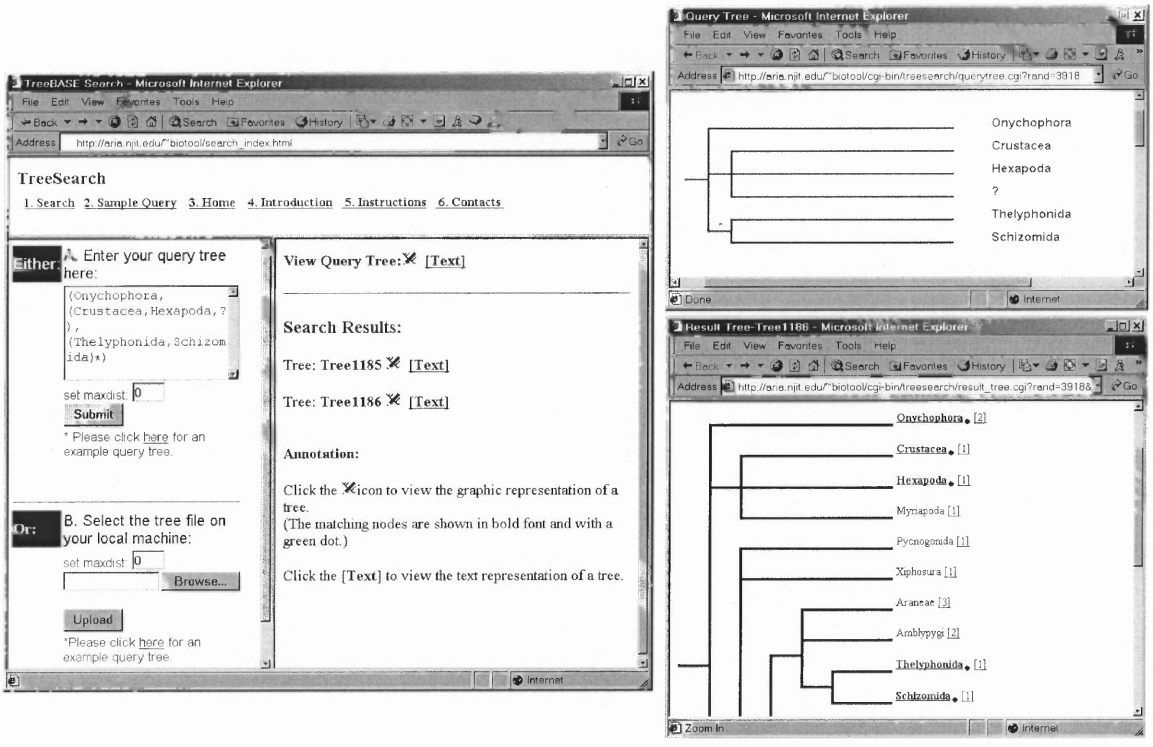


Figure 3.19 An example query and search results in the structure-based search engine for TreeBASE.

its local memory. First the master processor filter all the data trees. The candidate trees are kept in a global queue structure.

The master processor keep track of the global candidate queue. Each time it allocates a new piece of work to an idle processor until the global candidate queue is empty. As shown in Figure 3.18, LB-ATreeGrep outperforms both ATreeGrep and PB-ATreeGrep. The running time of ATreeGrep on one processor, Partition-Based ATreeGrep on 4 processors and Load Balancing ATreeGrep on 4 processors are compared . Figure 3.18 shows that both PB-ATreeGrep and LB-ATreeGrep improve the performance when more resources are available. The LB-ATreeGrep by taking advantage of the idle time among all processors, runs faster than PB-ATreeGrep.

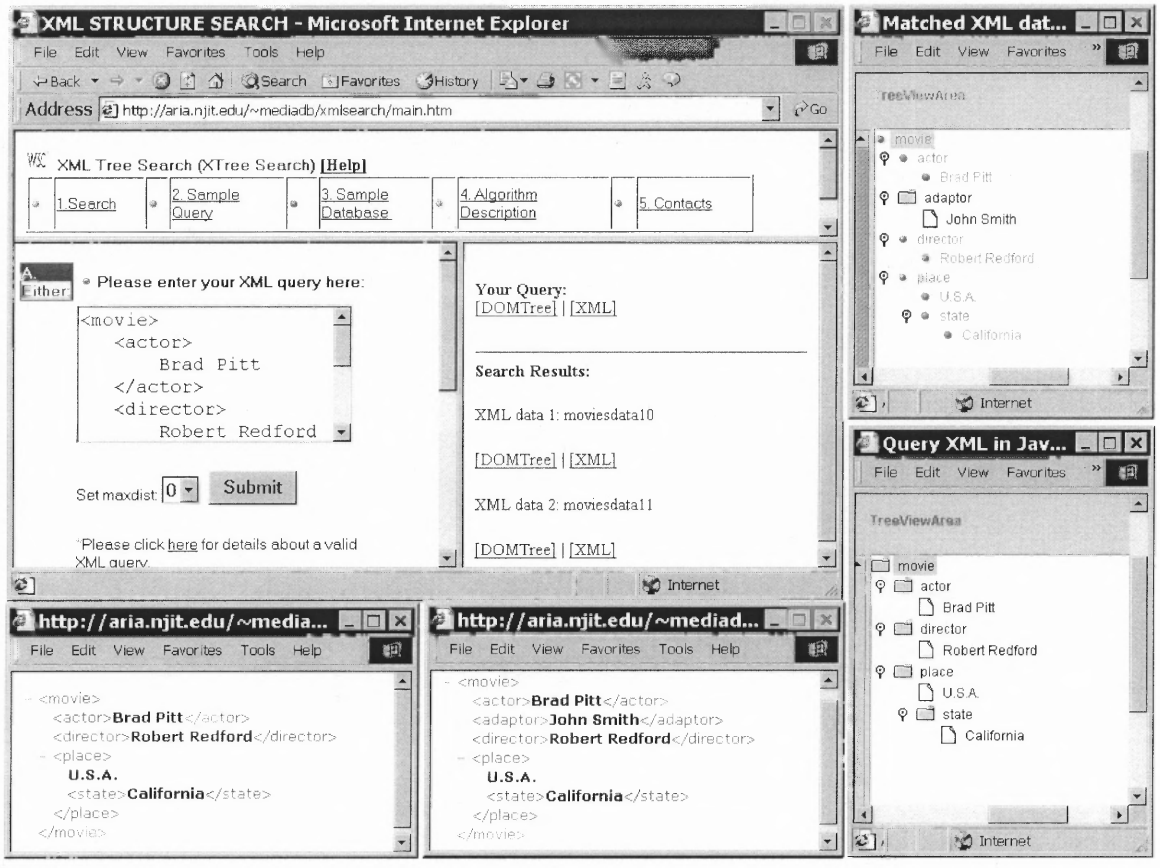


Figure 3.20 An example query and search results on a movie document database in XML QBE.

3.5 Applications

ATreeGrep has been incorporated into two Web-based systems. The first one is a structure-based search engine[53], accessible at `http://aria.njit.edu/~biotool/`, and now incorporated into TreeBASE's keyword-based search engine, accessible at `http://www.treebase.org/treebase/console.html`. This structure-based search engine is visited a few hundred times a month. Figure 3.19 shows its querying interface (in the left window), a query tree (in the right, top window) and a data tree (in the right, bottom window). In the figure, the query tree matches the data tree with distance 0, "?" matches "Myriapoda" and "*" matches a path of the data tree. This query finds all the phylogenetic trees in TreeBASE containing the query tree. Basically this structural search allows one to specify the relationship between taxa.

Allowing don't care symbols further enhances the power of the query language, and offers more flexibility to the structural search. The string notation employed in the input box of the left windows give a description of the topology. The graphical representation is shown in the the right, top window. For example, the query tree Q in figure 3.12 can be represented as a string "((e)?,((c)b)*)a".

The second system that has been implemented, called XML Query by Example (or XML QBE), allows the user to input an example XML fragment (query tree) and then finds those XMLs in an XML database that approximately contain the query tree. For example, the query in Figure 3.20 is to find all the XML documents describing movies in which Robert Redford is the director, Brad Pitt is an actor, and the movies are made in California, U.S.A. Shown in the figure are (counterclockwise, starting from upper left) the main menu, the querying window, the example XML (query) displayed via a Microsoft IE browser, a matching XML containing the query displayed via the IE browser, the query tree displayed via Java tree show applets, and the matching XML tree displayed via Java tree show applets. The matched portions in the matching XML tree are highlighted and marked with a bullet.

In general, when interacting with XML QBE, the user is able to type in his own query, load the query from a file, or use and modify a sample query provided by the system. The user is also able to browse the underlying database and read the XML documents in the database.

Molecular similarity is one of the major concepts in the design of new drugs. The molecule can be described by a free tree structure representing its major chemical building blocks and the way they are connected [51]. The 2D representation of chemical molecules, as described in such as the Merck Index,comprises cycles (e.g. benzene rings) and atoms. In applying the free tree search algorithm to do substructure search, the feature tree's representation of molecules is adapted. Figure 3.21 shows a pattern molecule P, a data molecule$D$.

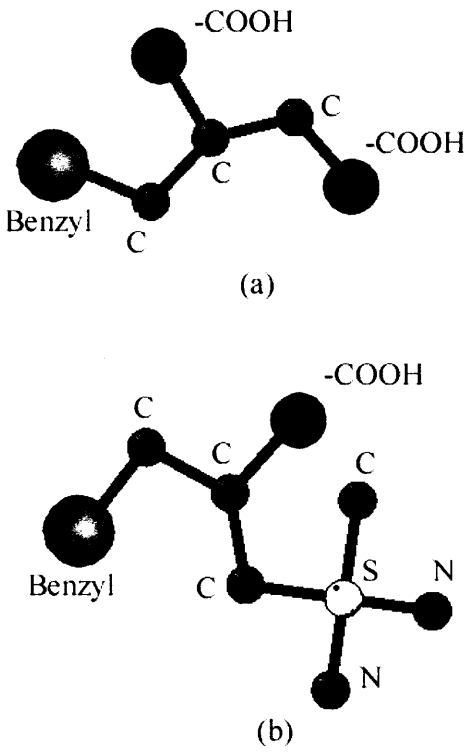


Figure 3.21 A pattern molecule P and a data molecule D and a path distance from tree P to tree D.

In figure 3.21, D approximately contains P with distance 1. In general, a pattern can be a base structure comprising one or two functional groups. ATreeGrep is applied to search a database of molecules to find those approximately containing the pattern. The result is a collection of molecules with the desired property, which can be used for further chemical analysis or drug design.

CHAPTER 4

TREERANK: A SIMILARITY MEASURE FOR NEAREST NEIGHBOR SEARCH IN TREEBASE

4.1 Introduction

Scientists model phylogenetic relations using unordered labeled trees and develop methods for constructing these trees [2, 8, 32, 33].[1] Different theories concerning the phylogenetic relationship of the same set of species often result in different phylogenetic trees. Even the same phylogenetic theory may yield different trees for different orthologous genes. With the unprecedented number of phylogenetic trees constructed based on these various theories, the need to analyze the trees and manage phylogenetic databases is urgent and great [44]. One important problem in this domain is to be able to compare the trees, thus possibly determining how much two theories have in common [6, 18, 23, 34]. The common portion of two trees may represent the actual phylogenetic relationship of the corresponding species.

The motivation for studying the tree matching problem comes from the design of tools for analyzing the phylogenetic data. One particular tool developed is a system for searching phylogenetic trees. Given a query or pattern tree P and a set of data trees $\mathcal{D}$, this search engine is able to find and rank nearest neighbors of P in $\mathcal{D}$. The importance of nearest neighbor searching for trees, particularly in phyloinformatics, has been addressed in the literature [53, 54, 55, 62]. Central to the search engine is an algorithm for computing the similarity score from P to each data tree D in $\mathcal{D}$.

The data consists of the phylogenetic trees stored in the widely used phylogenetic information system TreeBASE [50, 52], accessible at `http://www.treebase.org`.[2]

[1] These trees could be rooted ones, or could be unrooted ones, i.e. free trees. This chapter focuses on rooted, unordered phylogenetic trees.

[2] TreeBASE is a joint research effort developed at Harvard, UC Davis, Leiden University and the University at Buffalo.

Existing algorithms are mainly concerned with constructing the phylogenetic trees and finding the consensus of two trees, as opposed to information retrieval and ranking. By taking advantage of the properties of the phylogenetic trees and by utilizing the additive distance matrix [7] widely adopted in phylogenetic analysis, a new similarity measure, called *TreeRank*, is proposed for comparing the trees and for retrieving and ranking these trees while performing nearest neighbor searches in the phylogenetic database.

4.2 Phylogenetic Trees

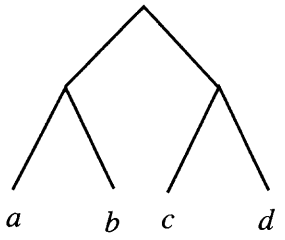
In general, phylogenetic trees are structures used in biology to model the evolution of various life forms and thereby the relationship of a particular life form with other life forms. The currently existing life forms or organisms usually appear as leaf nodes in these trees.[3] Each internal node of one such tree represents an inferred ancestor organism of the organisms represented by its child nodes. There can be multiple levels of ancestors, with multiple organisms sharing the same ancestors.

For the phylogenetic trees in TreeBASE (and for those generated by any of the modern programs), the following properties hold:

- each leaf node has a label and that label appears only once in the tree, though it may appear in other trees;
- each non-leaf node either has a label that appears nowhere else in the tree or has no label;
- each unlabeled internal node has at least two children. An unlabeled internal node stands for an extinct species from which new species branched out.

Additive distance[2, 7, 21] is widely used in phylogeny analysis. Given a two-dimensional additive distance matrix M in which each entry $M[u, v]$ is an integer, a

[3]More precisely, an organism (species or taxon) name appears as a label of a leaf node in a tree.



	a	b	c	d
a	0	2	4	4
b	2	0	4	4
c	4	4	0	2
d	4	4	2	0

Figure 4.1 An additive distance tree and its distance matrix.

free tree T is called an *additive distance tree* with respect to M if, for every pair of labeled nodes (u, v) in T, the path connecting node u and node v has exactly $M[u, v]$ edges.[4] Figure 4.1 shows an additive distance matrix and its corresponding additive distance tree. In the figure, for example, $M[a, d] = M[d, a] = 4$ meaning that there are four edges in the path connecting nodes a and d.[5] In general, each edge in an additive distance tree may be associated with a weight. In that case, $M[u, v]$ equals the sum of weights of the edges in the path connecting u and v. Given an additive distance matrix, many programs available on the Web can be used to reconstruct its phylogenetic tree [58].

In the following section by modifying the additive distance matrix, an *UpDown matrix* and a new distance measure, called *UpDown distance*, is proposed for comparing two rooted unordered phylogenetic trees.

4.3 UpDown Distance

4.3.1 Up and Down Operations

As stated above, rooted unordered phylogenetic trees satisfying the three properties described in Section 2 are discussed here, and these trees are simply referred to as

[4]A free tree is an unrooted unordered tree, which is also known as an undirected acyclic graph [60, 69].

[5]In this chapter, a node is often referred to by the label of that node and vice versa when the context is clear.

trees when the context is clear. two types of operations, up and down, between any two nodes in a tree are considered . These operations are intended to capture the hierarchical structure in the tree (reminiscent of the "up" and "down" operations used to define the partial order on pairs of nodes in [35]). If v is a child node of u, $v \uparrow u$ is used to represent an *up* operation from v to u, and use $u \downarrow v$ to represent a *down* operation from u to v. Then, for any pair of nodes m, n in the tree T, one can count the number of up and down operations to move, say a token, from m to n.

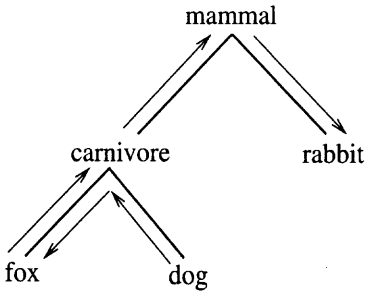


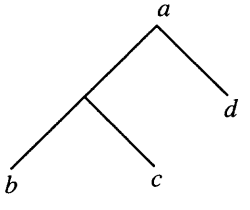
Figure 4.2 Illustration of up and down operations between two nodes in a tree.

For example, consider the tree in Figure 4.2 and the two nodes "fox" and "rabbit" in the tree. It takes two up operations ("fox $\uparrow$ carnivore" and "carnivore $\uparrow$ mammal") and one down operation ("mammal $\downarrow$ rabbit") to go from "fox" to "rabbit" in the tree. As another example, it takes one up operation ("dog $\uparrow$ carnivore") and one down operation ("carnivore $\downarrow$ fox") to go from "dog" to "fox" in the tree.

4.3.2 UpDown Matrix

Given a tree T, two matrices, referred to as the *Up matrix* U and the *Down matrix* D can be built , of integer values where $U[u, v]$ represents the number of up operations from node u to node v in T and $D[u, v]$ represents the number of down operations from u to v in the shortest path. Obviously $U[u, u] = D[u, u] = 0$ for any node u in T.

Figure 4.3 shows a tree and its Up and Down matrices. Notice that one of the internal nodes, namely the parent of b and c, does not have a label. The unlabeled node does not appear in the matrices. Comparing the Up and Down matrices with an additive distance matrix, cf. Figure 1, The main difference is that the additive distance matrix M is built for an unrooted unordered tree, i.e. a free tree. Each entry $M[u, v]$ represents the number of the undirected edges on the path connecting u and v. Thus, M is symmetric and $M[u, v] = M[v, u]$. By contrast, the Up and Down matrices are built with respect to a rooted tree and take into account the up and down operations along the directed edges between two nodes. Consider, for example, the nodes d and b in the tree T in Figure 4. If this tree were treated as an unrooted free tree, the number of edges between d and b would be 4. However, $U[d, b] = 1$ (representing 1 up operation, $d \uparrow a$) and $D[d, b] = 2$ (representing 2 down operations, $a \downarrow \lambda$ and $\lambda \downarrow b$ where λ represents the unlabeled node in T). Notice also that in this example, $U[b, d] = 2$ which is not equal to $U[d, b]$. Likewise $D[d, b] \neq D[b, d]$.



T

(i)

	a	b	c	d
a	0	0	0	0
b	2	0	1	2
c	2	1	0	2
d	1	1	1	0

U

(ii)

	a	b	c	d
a	0	2	2	1
b	0	0	1	1
c	0	1	0	1
d	0	2	2	0

D

(iii)

Figure 4.3 A tree and its Up and Down matrices.

The following are some facts that can be observed directly from the above definitions.

Fact 4.1 *For any pair of nodes u, v in a tree T, $U[u, v] = D[v, u]$.*

Hence, matrix D can be obtained from matrix U, and vice versa. therefore only matrix U is used throughout the chapter and refer to it as the UpDown matrix. The UpDown matrix describes the structure of T.

Fact 4.2 *For each node u in a tree T, $U[u,u] = 0$.*

Fact 4.3 *Suppose u, v are two nodes in a tree T and u is the parent node of v. Then*
(i) $U[u,v] = 0$ and $U[v,u] = 1$;
(ii) For each node w in T, $w \neq u$, $w \neq v$, such that w is not a descendant of u, $U[w,v] = U[w,u]$ and $U[v,w] = U[u,w] + 1$.

Fact 4.4 *Suppose u, v are two nodes in a tree T and u is the parent node of v. Then*
(i) $U[u,v] = 0$ and $U[v,u] = 1$;
(ii) For each node w in T, $w \neq u$, $w \neq v$, such that w is not a descendant of u, $U[w,v] = U[w,u]$ and $U[v,w] = U[u,w] + 1$.

Proof. (i) is obvious, since there is no moving-up operation from the node u to its child node v and there is exactly one moving-up operation from v to u. For (ii), there are two sub cases to consider:

(a) w is an ancestor of u. In this case, $U[w,v] = U[w,u] = 0$ and $U[v,w] = U[u,w] + 1$.

(b) w and u have a least common ancestor, x, $x \neq w$, $x \neq u$. In this case, $U[w,v] = U[w,u] = U[w,x]$. Furthermore, $U[v,w] = U[v,x] = U[u,x] + 1 = U[u,w] + 1$.

Fact 4.5 *If $\exists V_x$, such that $U[i,x] \neq 0$ and $U[x,i] = 0$ then $\forall V_k \in V, V_k \neq V_i$ and $V_k \neq V_x$,*

$$U[x,k] = \begin{cases} U[i,k] - U[i,x] & if(U[i,k] \geq U[i,x]) \\ 0 & otherwise \end{cases}$$

and

$$U[k,x] = \begin{cases} U[k,i] & if(U[i,k] \geq U[i,x]) \\ U[i,x] - U[i,k] + U[k,i] & otherwise \end{cases}$$

Proof. For a tree structure, it is easy to see that there is only one path from the root to one of the leaf nodes. Note that $U[i,x] \neq 0$ and $U[x,i] = 0$, which means that there is some moving-up operations form V_i to V_x and no moving-up operation from V_x to V_i, so V_x must be an ancestor of V_i. For each node $V_k \in V, V_k \neq V_i$ and $V_k \neq V_x$, let V_c be the least common ancestor of V_k and V_i. There are 4 cases to be considered.

Case 1: $V_c = V_i$. Both V_i and V_x are ancestors of V_k. By definition $U[i,k]=0$, and $U[i,x] > 0$, so $U[i,k] < U[i,x]$. Because V_x is an ancestor of V_k, $U[x,k]$ must be 0. The moving-up operations from V_k to V_x must be equal to the total moving-up operations from V_k to V_i then from V_i to V_x. Thus $U[k,x] = U[k,i] + U[i,x]$. By definition $U[i,k]=0$.

Case 2: $V_c = V_x$. The path from V_i to V_k must via V_x, and V_x is an ancestor for both V_i and V_k, which means $U[i,k] = U[i,x]$. So $U[x,k] = U[i,k] - U[i,x]=0$. It is easy to see that $U[k,i]=U[k,x] + K[x,i]$. Here $K[x,i]=0$, thus $U[k,x] = U[k,i]$.

Case 3: $V_c \neq V_x$, V_c is an ancestor for both V_x and V_i. Note that V_x is an ancestor of V_i. The number of moving-up operations from V_i to V_k must be greater than the total moving-up operations from V_i to V_x, so in this case $U[i,k] > U[i,x]$. Obviously $U[k,x] = U[k,i]=0$ and $U[x,k] = U[i,k] - U[i,x]$.

Case 4: $V_c \neq V_i$, V_x is an ancestor for V_c, and V_c is an ancestor of V_i. By definition, $U[i,k] < U[i,x]$ and $U[x,k]=0$.

(i) $U[k,x]=U[k,c] + U[c,x]$

(ii) $U[k,c]=U[k,i]$

(iii) $U[c,x]=U[i,x]$ -$U[i,k]$

From (i), (ii) and (iii), $U[k,x]$=$U[i,x]$ - $U[i,k]$ + $U[k,i]$.

Given an $m \times m$ UpDown matrix U consisting of only the labeled nodes, by applying Fact 4.4, an UpDown matrix U' consisting of all nodes can be obtained. Given an $n \times n$ UpDown matrix U', its corresponding tree T can be reconstructed. Suppose there are k internal nodes that are unlabeled in T. these nodes are associated with $\lambda_1, \ldots, \lambda_k$ where λ_i, $1 \leq i \leq k$, is not in the label alphabet. There are two observations, cf. Figure 3:

- There must exist a row in U' that contains all zeros. This row corresponds to the root of T.
- For each node $v \in T$ where v is not the root, its row in U must have one and only one column, say u, where $U'[v,u] = 1$ and $U'[u,v] = 0$. In fact, this node u is the parent of v in T.

Thus, the tree reconstruction algorithm works by scanning the matrix U', from top to bottom, and constructing the parent-child pairs, which are then glued into T. Since the nodes in T are uniquely labeled and the unlabeled nodes are associated with a distinct λ_i, the algorithm produces a unique tree T. The time complexity of this algorithm is $O(n^2)$ where n is the number of nodes of the tree.

lemma 4.1 *Two trees are the same if and only if their UpDown matrices are the same.*

Figure 4.4 illustrates how to reconstruct a tree from its UpDown matrix, using the UpDonw matrix U in Figure 4(ii) as a running example. Since the row for a contains all zeros, a is identified as the root of the tree. Then for the λ in the second row of U, since $U[\lambda,a] = 1$ and $U[a,\lambda] = 0$, the parent-child pair a-λ is created as shown in Figure 4.4(i). By continuing scanning the remaining rows, the tree can be reconstructed shown in Figure 4.4(iv), which is the same as the tree in Figure 4.3(i).

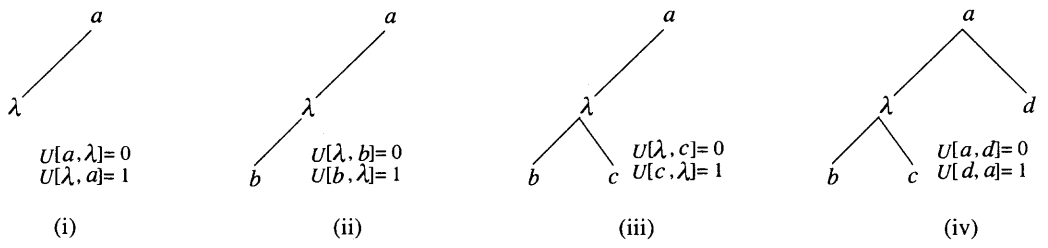


Figure 4.4 Illustration of constructing a tree from an UpDown matrix.

4.3.3 TreeRank

In general, when using a search engine, if the user inputs a query tree with three nodes "fox","dog" and "tiger" plus their parent node "mammal", the user often expects to see data trees in search results containing these nodes. If the user doesn't want to see a search result containing, for example, a node "tiger", he or she can simply input a query tree having "fox", "dog" and "mammal" only.

This implies that in designing a search engine and similarity measure, the following two criteria should be considered together:

1. whether all, or at least most of, the labeled nodes of the query tree P occur in a data tree D;

2. to which extent the query tree P is similar to the data tree D in structure.

These criteria should be considered when seeking nodes in D that match nodes in P when comparing P with D. Specifically, let V_P be the set of labeled nodes in P and let V_D be the set of labeled nodes in D. Let U_P represent the UpDown matrix of P and let U_D represent the UpDown matrix of D. Let $Intersect(P, D)$ denote the intersection of V_P and V_D and let $Difference(P, D)$ denote $V_P - V_D$. The *UpDown*

distance is defined from P to D, denoted $UpDown\_dist(P, D)$, as

$$Updown\_dist(P, D) = \sum_{u \in I} \sum_{v \in I} |U_P[u, v] - U_D[u, v]| + \sum_{u \in J} \sum_{v \in J} U_P[u, v] \quad (4.1)$$

The TreeRank score from P to D, denoted $TreeRank(P, D)$ or $USim(P, D)$, is calculated by

$$TreeRank(P, D) = (1 - \frac{UpDown\_dist(P, D)}{\sum_{u \in V_P} \sum_{v \in V_P} U_P[u, v]}) \times 100\% \quad (4.2)$$

The Treerank score from P to D is a measure of the topological relationships in P that are found to be the same or similar in D. If P and D are the same or if one can find a substructure in D that exactly matches P, then $TreeRank(P, D) = 100\%$. On the other hand, if P and D do not have any labeled node in common, then $TreeRank(P, D) = 0$. The time complexity of the algorithm for computing $TreeRank(P, D)$ is $O(M^2 + N)$ where M is the number of nodes appearing in both P and D, and N is the number of nodes in D.

4.4 Nearest Neighbor Searching

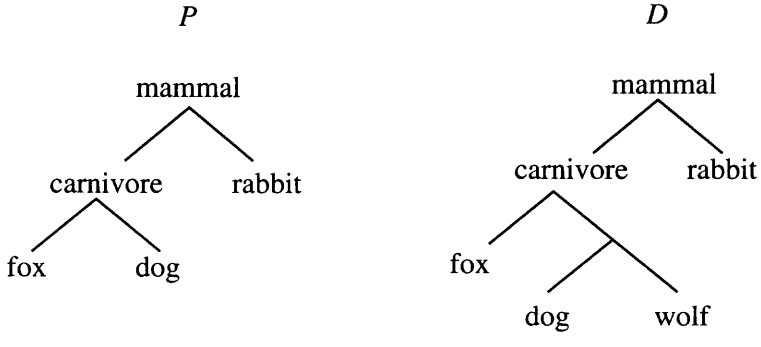


Figure 4.5 Example trees.

Figure 4.5 shows a query tree P and a data tree D that satisfy the three properties described in Section 2. In the biological sense, when comparing P with D,

their distance should be 0. To solve distance of this kind of trees, a *data tree reduction* technique is incorporated into the nearest neighbor searching algorithm, which works as follows.

Consider a query tree P and a data tree D and their UpDown matrices. Find the column and row indexes of the nodes in the intersection of V_P and V_D. Mark those matching nodes in D with asterisks. If two distinct nodes of D are marked, then their least common ancestor is also marked. The reduced data tree D' of D contains only the marked nodes. Equivalently, unmarked nodes having only one neighbor (this must preserve connectedness) are remove. The above removal might yield additional unmarked nodes with one neighbor, which themselves will be removed. If an unmarked node n is connected to two other nodes m_1 and m_2, then remove n and link m_1 and m_2. This too preserves connectedness. Continue doing these two operations until neither can be done. The node removal operation is similar to the "degree-2 delete" operation defined in [69] where a node can be deleted when the node's degree is less than or equal to 2. Notice that after reduction, the UpDown matrices will change, and the new matrices is used to calculate the similarity score of P and D.

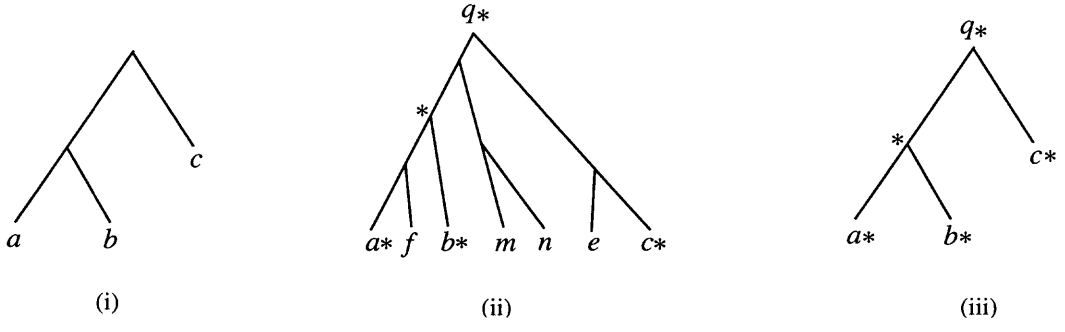


Figure 4.6 Example showing how the data tree reduction technique works in nearest neighbor searching.

Figure 4.6 presents an example. In the figure, (i) shows a query tree, (ii) shows a data tree in which some nodes are marked, and (iii) shows the reduced tree of the

data tree in (ii). In performing nearest neighbor searches, the algorithm first applies the tree reduction technique to a data tree D, and then calculates the TreeRank score from the given query tree P to the reduced tree of D using the formula described in Section 4.3. The resulting score is then presented as the similarity score from P to D.[6] For example, in Figure 4.6, since the TreeRank score from the query tree in (i) to the reduced data tree in (iii) is 100%, The algorithm displays the data tree in (ii) with a 100% similarity score to the query tree. This matching technique yields a similar effect as tree matching with variable length don't cares [54, 55], though the proposed approach does not require the user to explicitly specify the don't cares in the query tree.

4.5 Extensions to Weighted and Unrooted Trees

Some tree reconstruction methods provide information to build a weighted tree where the weight on an edge represents the estimated evolutionary distance between the two nodes connected by the edge [48]. In extending the above approach for weighted trees, each up and down operation is associated with a weight that equals the weight of the corresponding edge. Instead of having $U[u, v]$ represent the number of up operations from node u to node v, $U[u, v]$ is used to represent the sum of weights associated with the up operations from u to v. Likewise, $D[u, v]$ is used to represent the sum of weights associated with the down operations from u to v. It can be shown that two weighted trees are identical if and only if their weighted Updown matrices are the same. The similarity score between two weighted trees is then calculated in the same way as in Equation (4.2).

Some phylogenetic tree reconstruction methods such as MP [27] and ML [25] may produce unrooted unordered trees, or free trees. An unrooted tree is one that specifies only kinship relationships among taxa without specifying ancestry

[6] A data tree D is considered to be a neighbor of P if P and D share at least one common leaf label. Otherwise D is not a neighbor of P and is not displayed as a search result.

relationships. The common ancestor of all taxa is unknown. Each edge in an unrooted tree can be weighted or unweighted. Let T be an unrooted unordered tree. The *Additive matrix* A is defined for T where each entry $A[u, v]$ is the sum of the edge weights on the shortest path connecting u and v in T. If T is not weighted, then $A[u, v]$ is simply the number of edges on the shortest path connecting u and v in T (reminiscent of the additive distance for an unrooted tree described in [2, 7, 64]).

Now let A_P represent the Additive matrix of the query tree P and let A_D represent the Additive matrix of a data tree D. Let V_P be the set of labeled nodes in P and let V_D be the set of labeled nodes in D. Let I be the intersection of V_P and V_D; let J denote $V_P - V_D$. The Additive distance from P to D is defined, denoted $Add\_dist(P, D)$, as follows:

$$Add\_dist(P, D) = \sum_{u \in I} \sum_{v \in I} |A_P[u, v] - A_D[u, v]| + \sum_{u \in J} \sum_{v \in J} A_P[u, v] \tag{4.3}$$

The similarity score from P to D, denoted $ASim(P, D)$, is calculated by

$$ASim(P, D) = (1 - \frac{Add\_dist(P, D)}{\sum_{u \in V_P} \sum_{v \in V_P} A_P[u, v]}) \times 100\% \tag{4.4}$$

The time complexity of the algorithm for computing $ASim(P, D)$ is $O(M^2 + N)$ where M is the number of nodes in P, and N is the number of nodes in D. It can be shown that for two unrooted trees P and D, whether they are weighted or unweighted, P and D are identical if and only if the similarity score from P to D is 100%. This property holds for rooted trees as well.

4.6 A Filter

Given a query or pattern tree P and a database of phylogenies $\mathcal{D}$, the goal is to find near neighbors of P in $\mathcal{D}$ where the similarity scores between the near neighbors and P are greater than or equal to a user-specified threshold δ. A filter is developed to speed up the search, which works as follows. For the database of trees, a hash table keyed by pair of node labels and each hash bin contains tree identification numbers is created. The pair can be in alphabetical order because $U[u, v] = D[v, u]$ for any pair of node labels (u, v). Now given the query tree P, each pair of node labels in P is considered. Which trees of the database the pair is in is checked. (This requires time independent of the size of the database.) Sort the data trees by the number of hits.

By evaluating a data tree D, a lower bound on the Updown distance from P to D can be got by looking at $U_P[u, v]$ where U_P is the Updown matrix of P and (u, v) is a pair in P that is missing from D. The lower bound, denoted *Low*, is computed by summing up $U_P[u, v]$ for all pairs of (u, v) of P that are missing from D. From the lower bound, an upper bound, denoted *Upp*, can be calculated on the similarity score from P to D, where

$$Upp = (1 - \frac{Low}{\sum_{u \in V_P} \sum_{v \in V_P} U_P[u, v]}) \times 100\% \tag{4.5}$$

and V_P is the set of labeled nodes in P.

If the upper bound is already smaller than the user-specified value δ, D can be eliminated from consideration without calculating the similarity score from P to D. For example in Figure 4.7, ϵ is set to 5. The sums of the missing pairs in T_1 and T_2 are 6 and 8 respectively, so tree T_1 and T_2 are discarded for further consideration.Furthermore, if a data tree D has a set S of k hits and it is decided D doesn't qualify to be a solution after calculating the similarity score from P to D,

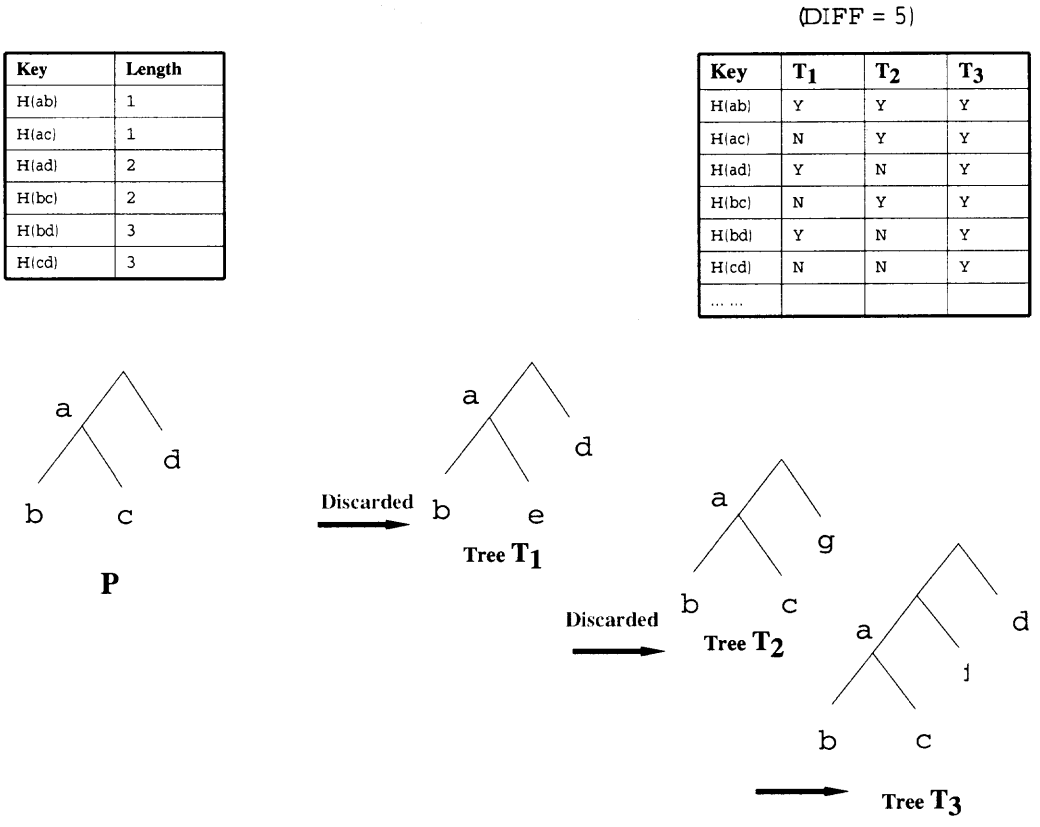


Figure 4.7 Filter examples.

then any data tree D' that only has S' of k' hits, where $k' < k$ and S' is a subset of S, will not be a solution and hence can be eliminated from consideration. As the experimental results show later, this filtering technique works well in practice.

1548 phylogenetic trees in TreeBASE are considered. Most non-leaf nodes in these trees have two children. The number of nodes of a tree ranges from 50 to 200, and the dictionary size of node labels is 18870. Figure 4.8 compares TreeRank and TreeRank with filter for varying query tree sizes. The figure shows that filter speeds up TreeRank considerably. This filter helps to eliminate, on average, more than 95% of data trees in performing nearest neighbor searches in TreeBASE.

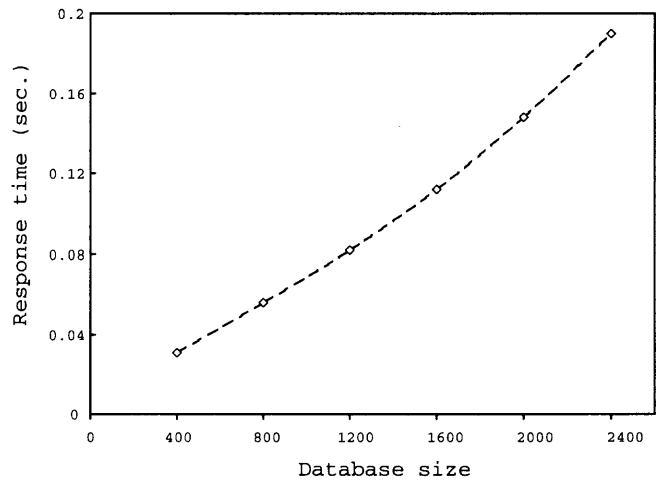


Figure 4.8 Running times of TreeRank and TreeRank with filter on the 1548 phylogenetic trees obtained from TreeBASE.

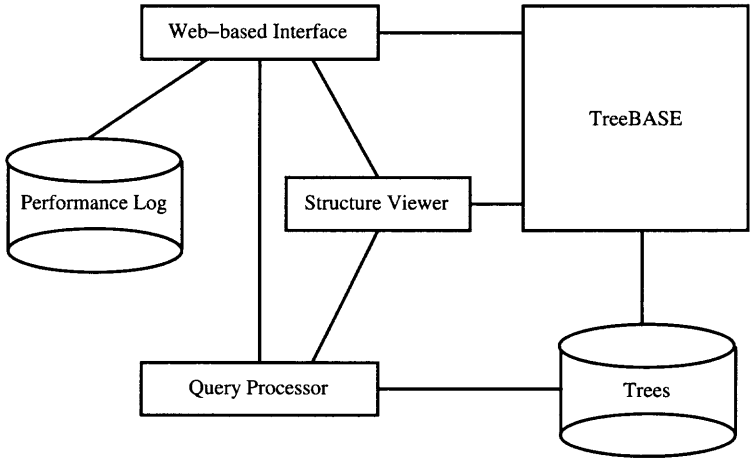


Figure 4.9 The software architecture of the proposed search engine.

4.7 Implementation

The nearest neighbor searching algorithm proposed here has been incorporated into a Web-based system. The search engine is implemented using Java, HTML, Perl CGI, and C. It is fully operational and is accessible at `http://aria.njit.edu/~biotool/treerank.html`. Figure 4.9 shows the software architecture of the search engine. The system is comprised of four components: Web-based Interface, Query Processor, Structure Viewer and Performance Log. From Web-based Interface, the user is able to type in his/her own query (an example tree), upload the query tree from a file, or use and modify a sample query provided by the system. Query Processor searches TreeBASE for phylogenetic trees that are nearest neighbors of the query tree using the algorithm described in Section 4.4. Structure Viewer displays the trees using either a parenthesized string notation or a dendrogram format, which are presented to the user via Web-based Interface. User queries and their time-stamps are maintained in Performance Log, which helps to analyze user needs and better tune the system for working more effectively. The search engine is connected to TreeBASE on the Web and therefore it uses the visualization tools available in TreeBASE for displaying trees graphically.

Figure 4.10 shows the system's main screen and query interface (the upper left window), a query tree (the lower left window), and the query tree's nearest neighbor in TreeBASE (the right window). In the main screen, the query tree is expressed in the parenthesized string notation; in the other two windows this same query tree and the nearest neighboring tree are viewed in the dendrogram format. In general, to view a tree in the dendrogram format, the user would need to click the icon with the pencil overlaid upon the phylogenetic tree in the main screen. To view the parenthesized string notation, the user would need to click on the "Text" link in the main screen.

Figure 4.10 shows that Tree1411 is ranked highest, which is the nearest neighbor of the query tree with a 100% similarity score. Other similar trees are also displayed

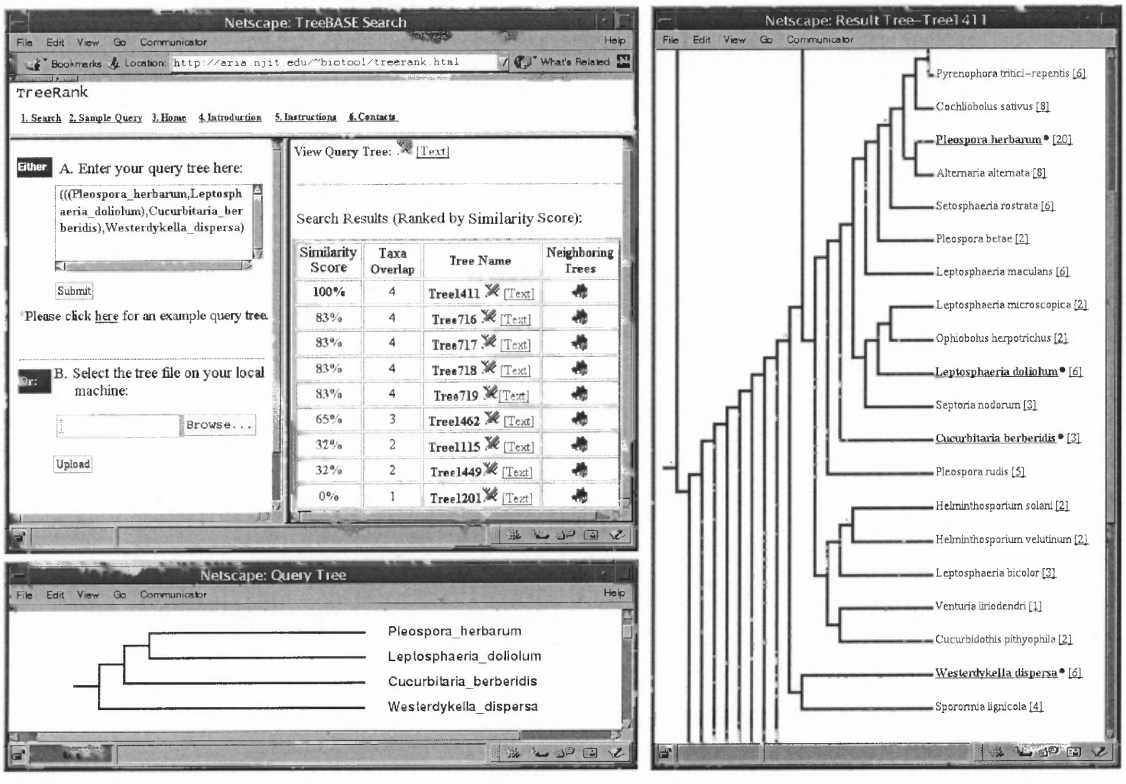


Figure 4.10 An example query and search results displayed via the Web-based interface of the proposed search engine.

and ranked, from top to bottom, based on their similarity scores. For each displayed data tree D, when clicking on the icon in its "Neighboring Trees" column, the system will initiate a new search using D as the query tree, and rank and display D's nearest neighbors in TreeBASE. The "Taxa Overlap" column shows the number of leaves (taxa) D has that also appear in the query tree. The system only displays similar trees whose "Taxa Overlap" column has a number greater than 0.[7] Notice that even though the top five ranked data trees all have the same "Taxa Overlap" value, namely 4, only Tree1411 has a similarity score of 100%. The other four trees only have a similarity score of 83%.

Referring to the right window in Figure 4.10, the entire Tree1411 is drawn and the user can use scroll bars to view portions of the tree. Named clades (labeled internal nodes) are indicated by a red dot. Each leaf node (taxon, species, or organism) has a number next to its label. This number represents the number of studies in TreeBASE the taxon is found within. If the user clicks on this number, he or she is linked to TreeBASE so that he or she can search on that taxon about those studies. The specific taxa specified within the query tree are highlighted in Tree1411 using underscored red font with a green circle next to the taxon's name. It should be pointed out that after applying the tree reduction technique to Tree1411, the reduced tree is exactly the same as the query tree. Consequently the similarity score for Tree1411 is 100%.

4.8 Related Work

In the past, a number of similarity and distance measures for phylogenetic trees have been proposed. Various algorithms for tree matching [1, 43, 62, 68] and for tree reconstruction [2, 17, 35, 65] have been studied. Different theories, when applied

[7]If a data tree does not share any common taxon with the query tree, the algorithm filters out the data tree immediately without applying tree reduction to it and calculating its TreeRank score. This filter works well in practice. The benchmark result shows that for many query trees, the proposed system can perform a search on a set of approximately 1500 trees in about one second on a SUN Ultra 20 workstation.

to finding the phylogenetic relationship of the same set of species, often result in different phylogenetic trees. To determine how much two theories have in common is a fundamental problem in computational biology and in phyloinformatics.

The most comprehensive algorithmic and software tool in this field is perhaps the COMPONENT package (`http://taxonomy.zoology.gla.ac.uk/rod/cpw.html`) developed by Page at University of Glasgow. It provides several ways of finding the consensus of two phylogenetic trees. The first is to see whether they have similar "quartets" which are based on adjacency relationships among all possible subsets of four leaf species. The similarity is then computed as the proportion of quartets that are shared in the two trees [5, 6, 20].

Partition distance treats each phylogenetic tree as unrooted and analyzes the partitions of species resulting from removing one edge at a time. By removing an edge in a tree, one is able to partition that tree. The difference between two trees is defined as the number of edges for which there is no equivalent (in the sense of creating the same partitions) edge in the other tree [19].

The maximum agreement subtree between two phylogenetic trees T_1 and T_2 is a substructure of the two trees on which the two trees are the same [15, 16, 24, 37, 35, 38]. Commonly such a subtree will have fewer leaves than either T_1 or T_2. By contrast, a consensus tree has the same number of leaves as the original trees T_1 and T_2, assuming those trees have the same set of species. Consensus trees should be used gingerly, however, because a consensus tree is not a phylogeny unless the two trees are isomorphic. Instead, consensus trees are a convenient way to summarize the agreement between two or more trees. Consensus trees can be formed from cluster methods (strict, majority rule, semi-strict, or Nelson) or by intersection methods (Adams); see [19, 42, 45] for more details.

The last dissimilarity measure implemented in COMPONENT is the nearest neighbor interchange (NNI) distance. Given two unrooted unordered trees T_1 and

T_2 with the same set of labeled leaves, their NNI distance is the number of NNI operations needed to transform T_1 to T_2. Finding the NNI distance between two trees is NP-hard. Brown and Day [4] developed several approximation algorithms to calculate the distance, which are implemented in COMPONENT.

In contrast to the above distance and similarity measures, which are developed for comparing two trees, possibly with some constraints (e.g. the two trees must have the same set of leaves), TreeRank is mainly designed for nearest neighbor searching in phylogenetic databases. In [55], an approach called ATreeGrep is presented, which measures the distance between two general rooted unordered trees by counting the mismatching paths in the two trees. By utilizing a suffix array index structure, ATreeGrep focuses on fast retrieval in a database of general rooted unordered trees. The tool has been applied to processing XMLs [54] and phylogenies [53]. It allows the user to add variable length don't cares to a query tree when a certain portion of a data tree is unimportant or unknown in matching with the query tree. By contrast, TreeRank is specially designed for rooted phylogenetic trees that satisfy the three properties described in Section 2. It captures the structural difference of a data tree with respect to the query tree by considering the up and down operations in the two trees. The nearest neighbor searching algorithm proposed here employs the data tree reduction technique described in Section 4.4, without asking the user to explicitly specify variable length don't cares in the query tree.

4.9 Results

The filter technique has been tested on synthetic data. One thousand unweighted rooted trees were randomly generated, each tree having 100 nodes. The string labels of nodes were randomly chosen from a dictionary of size 500 . The threshold value δ was set to 60%. In each run, a tree was selected and modified into the query tree and the other trees were used as data trees. 1,000 runs were tested and the average

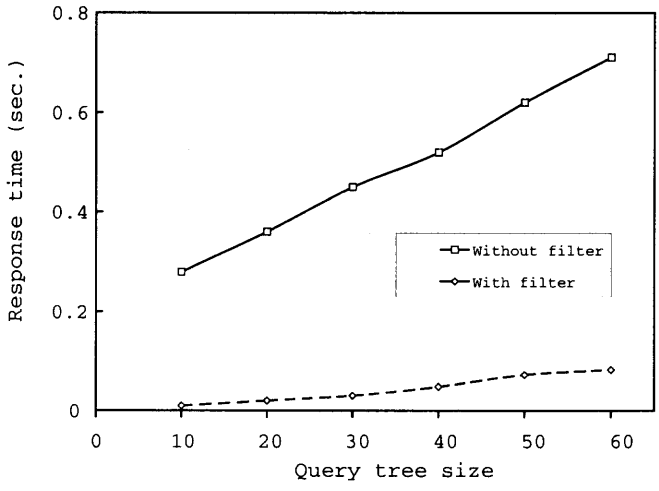


Figure 4.11 Running times on 1,000 synthetic trees for search methods with and without the filter.

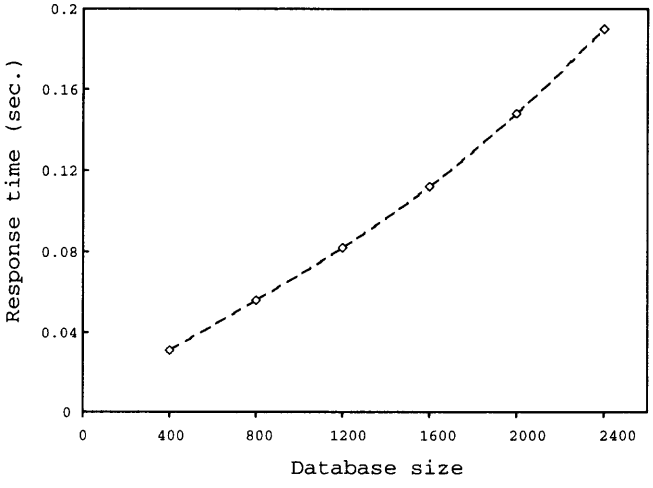


Figure 4.12 Running times of the proposed search method on different sizes of databases.

was plotted. Figure 4.11 shows the results for varying query tree sizes. It can be seen from the figure that the proposed filter speeds up searches considerably. It was also observed that the running time drops as the user-specified threshold value δ increases. This happens because fewer data trees survive the filter when δ becomes larger. Figure 4.12 shows that the proposed search method scales up well—its running time increases linearly with increasing number of trees. These results are consistent with those for real phylogenetic trees.

The proposed search method for unweighted rooted trees has been implemented into a Web-based system connected with TreeBASE. Figure 4.13 shows the system's main screen and query interface (the upper left window), a query tree (the lower left window), and the query tree's nearest neighbor in TreeBASE (the right window). In the main screen, the query tree is expressed in the parenthesized string notation; in the other two windows this same query tree and the nearest neighboring tree are viewed in the dendrogram format. Figure 4.10 displays data trees in TreeBASE where the similarity score, $USim$, of each data tree to the query tree is greater than or equal to the user-specified threshold, 60%. Among the data trees, Tree1411 is ranked highest, which is the nearest neighbor of the query tree with a 100% similarity score. It should be pointed out that after applying the tree reduction technique to Tree1411, the reduced tree is exactly the same as the query tree. (The matched taxa between the query tree and Tree1411 are highlighted with a bullet and underscored in the figure.) Consequently the similarity score for Tree1411 is 100%.

This structural search engine is implemented using Java, HTML, Perl, CGI, and C. It is fully operational and is accessible at `http://aria.njit.edu/~biotool/nnsearch.html`. As of December 2003, about 500 users worldwide have accessed the search engine over 7,000 times totally. Most submitted query trees are small trees with 20 or fewer nodes. With these query trees, a moderate similarity score (e.g. 60%),

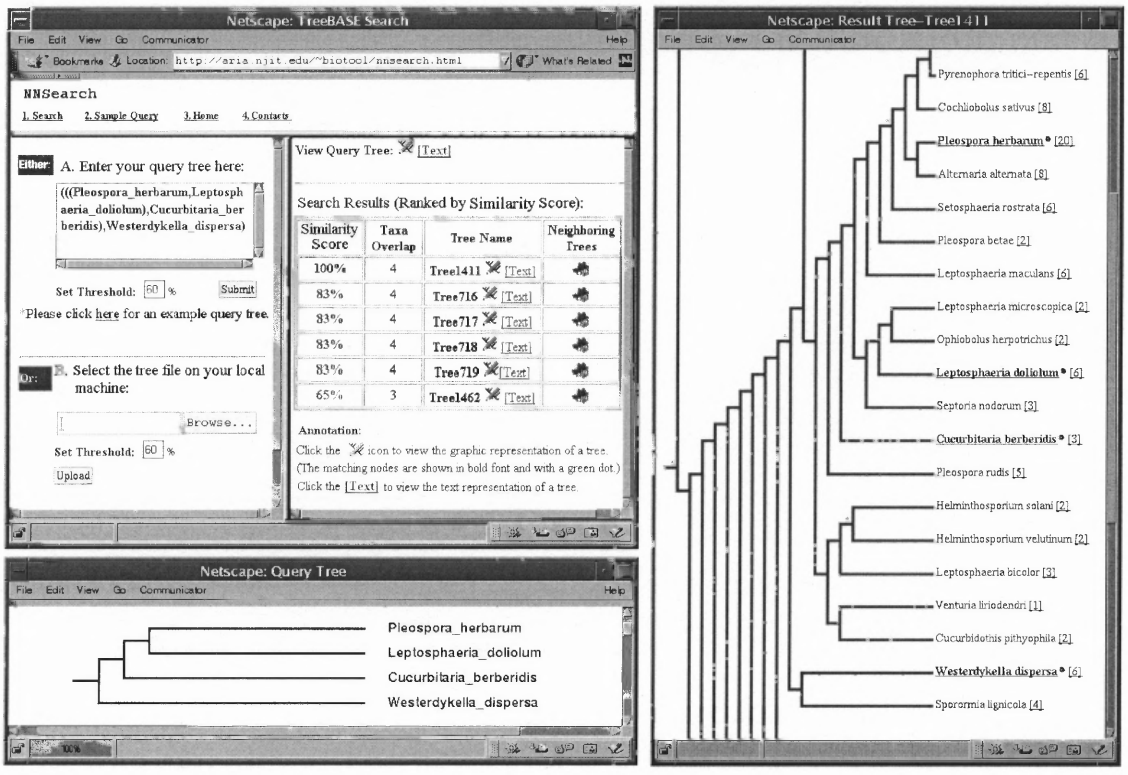


Figure 4.13 An example NN query and search results displayed via the Web-based interface of the proposed search engine.

and the approximately 1,600 unweighted rooted trees in TreeBASE, the system can perform a search in about one second on a SUN Ultra 20 workstation.

4.10 Discussion

Unlike many existing metrics [5, 4, 15, 19, 29, 35, 37, 38, 45, 47], designed for comparing two trees possibly with some constraints (e.g. the two trees must have the same set of leaves), the similarity scores described in the chapter are mainly developed for near neighbor searching in phylogenetic databases. The similarity scores are not symmetric, i.e. $USim(X,Y) \neq USim(Y,X)$, $ASim(X,Y) \neq ASim(Y,X)$, for any two trees X and Y. The non-symmetry property is good in query-driven phylogenetic information retrieval; it distinguishes between the situation in which X is a query and Y is a data tree and the situation in which Y is a query and X is a data tree.

Table 4.1 Comparison of the Five Studied Tree Metrics

Metric	Weighted trees	Internal labels	Unresolved trees	Different taxa	Polynomial computable
PAR	N	N	Y	N	Y
MAST	N	Y	N	Y	N
NNI	N	N	N	N	N
QUA	N	N	Y	N	Y
WSSP	Y	Y	Y	Y	Y

To evaluate the quality of the proposed similarity measures, *USim* is compared with four widely used tree metrics implemented in the COMPONENT tool [46]. These tree metrics include partition metric (PAR), nearest neighbor interchange metric (NNI), quartet metric (QUA) and maximum agreement subtree metric (MAST). Specifically, the distribution of the metric values on 945 unweighted rooted trees generated by the COMPONENT tool is compared. The query tree was generated randomly; the 945 data trees covered the entire tree space of unweighted rooted trees with 6 labels. The query tree is compared with each data tree to obtain a metric or (dis)similarity value. For MAST, the metric value equals the number of leaves removed to obtain a maximum agreement subtree of the query tree and the data tree. The results are summarized in Figures 4.14, 4.15, 4.16, 4.17 and 4.18. In each figure, the X-axis shows different metric values. For each specified value on the X-axis, the figure shows the number of data trees whose metric/(dis)similarity value from the query tree equals the specified value. An in-depth comparison between the four widely used tree metrics and the proposed similarity measures *USim* and *ASim*, collectively referred to as WSSP, is summarized in Table 1.

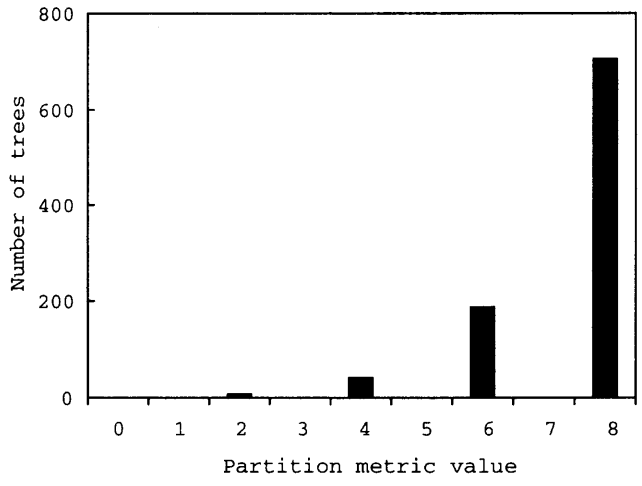


Figure 4.14 Distribution of PAR metric values.

It can see from Figures 4.14, 4.15, 4.16, 4.17 and 4.18 that WSSP gives a good distribution of values, unlike partition metric (PAR) and maximum agreement subtree metric (MAST). From Table 1it can be seen that the running time of WSSP is better than MAST and NNI (nearest neighbor interchange metric). WSSP can be applied to weighted trees and unweighted trees where trees can be fully resolved or unresolved. It can be used to compare two trees whose internal nodes have labels and whose leaves have different taxa as shown in Table 1. The bottom line is that WSSP could be a useful metric in addition to the other excellent ones available.

In summary, a new approach to near neighbor searching for phylogenetic trees has been presented . Given a query or pattern tree P and a database of trees $\mathcal{D}$, the proposed approach finds data trees D where the similarity score of P to D is greater than or equal to a user-specified threshold value. Similarity measures are developed for comparing rooted and unrooted trees where the trees can be weighted or unweighted. The proposed algorithms have been used for analyzing the structures of phylogenetic trees and for performing structure-based searches in TreeBASE.

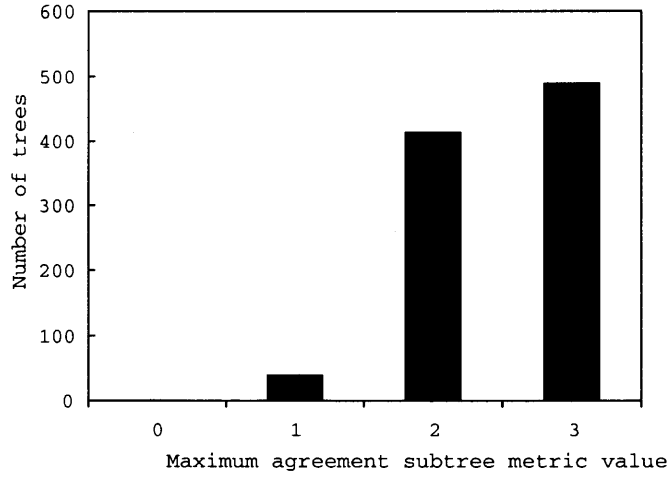


Figure 4.15 Distribution of MAST metric values.

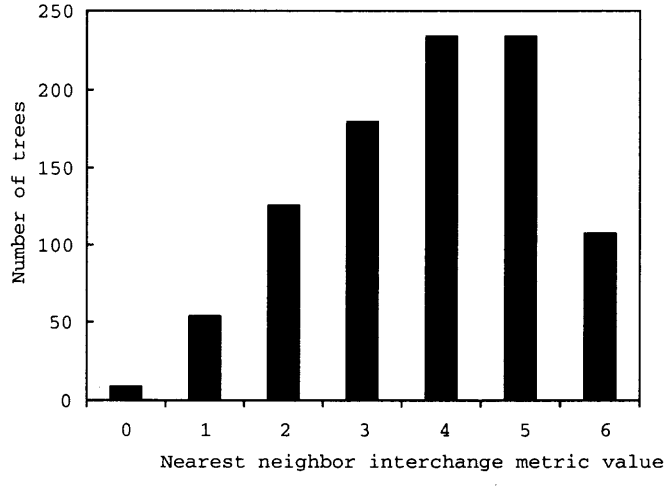


Figure 4.16 Distribution of NNI metric values.

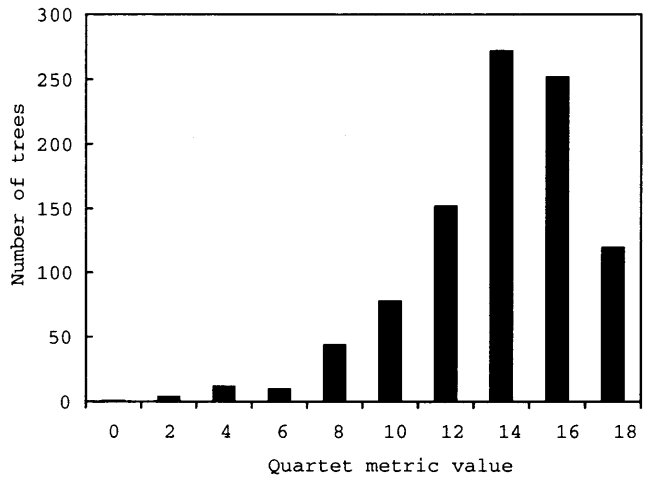


Figure 4.17 Distribution of QUA metric values.

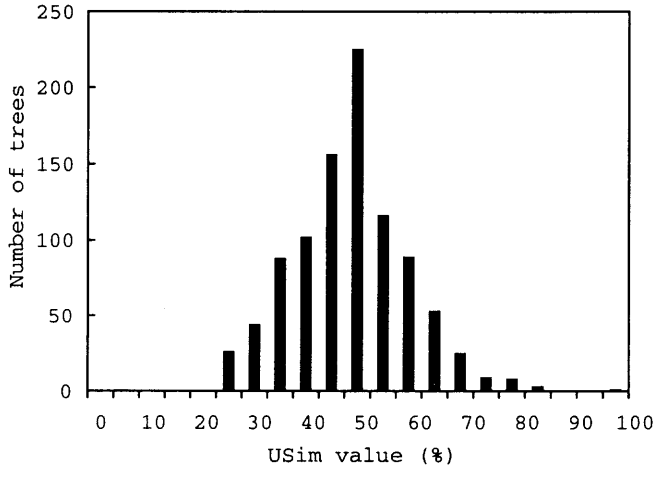


Figure 4.18 Distribution of *USim* values.

CHAPTER 5

WEB-BASED SYSTEM AND PROTOTYPES

TreeSearch is a research project conducted in New Jersey Institute of Technology, New York University and University of Western Ontario, Canada.

The project aims to produce algorithms, data structures, and tools that allow rapid and approximate search across phylogenetic trees. Currently the phylogenetic trees used are taken from TreeBASE. The underlying algorithms are based on Pathfix, Pathfilter and TreeRank.

This system allows users to perform various types of structure-based queries on TreeBASE. Users can click on the "Search" button in the menu to try this system. The "Instructions" button provides guidelines of using the system.

5.1 Screenshots

The main menu of TreeSearch is shown in Figure 5.1. The graphic representation of a query tree is shown in Figure 5.2.

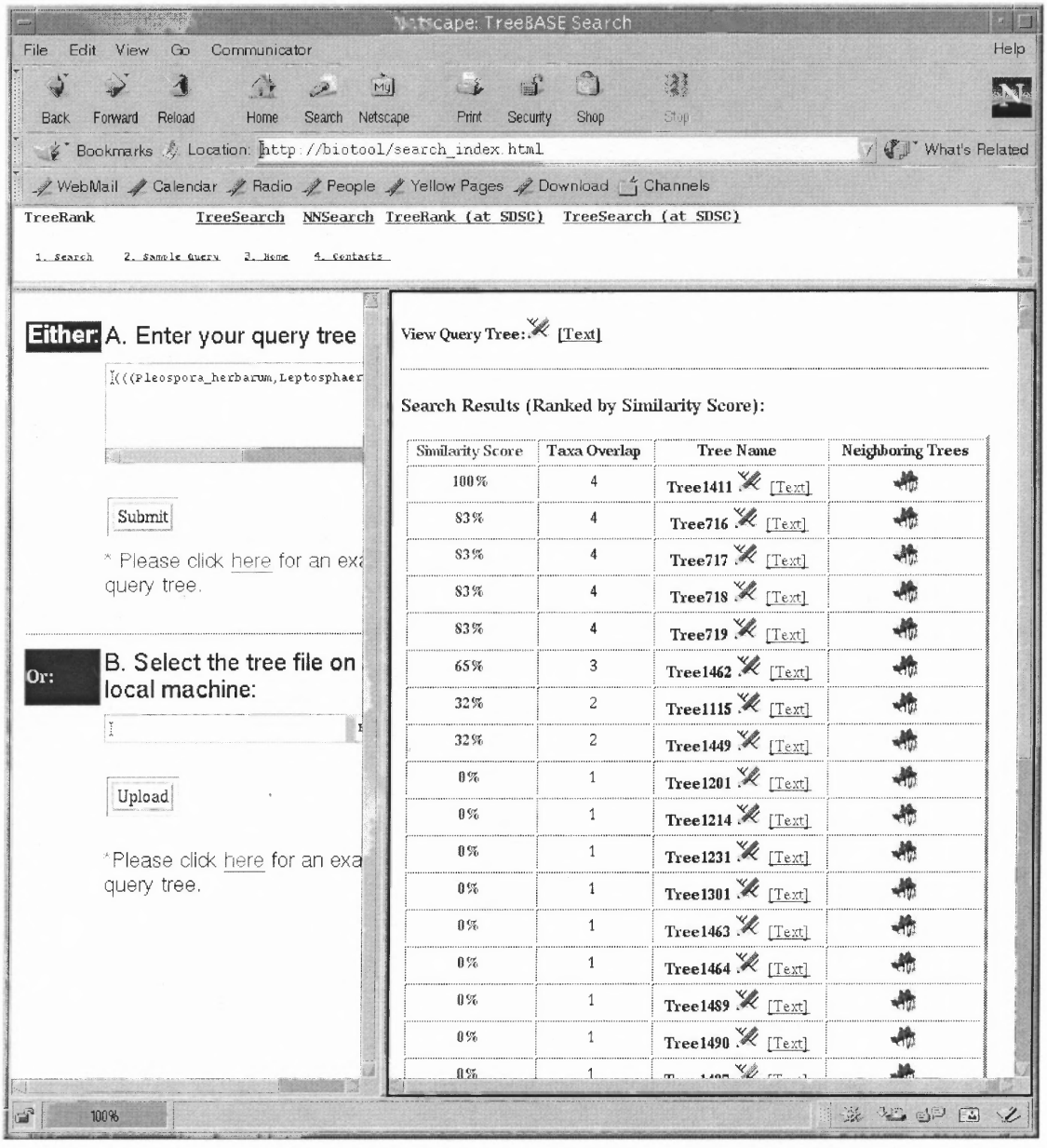


Figure 5.1 Main menu of the structural search engine on TreeBASE.

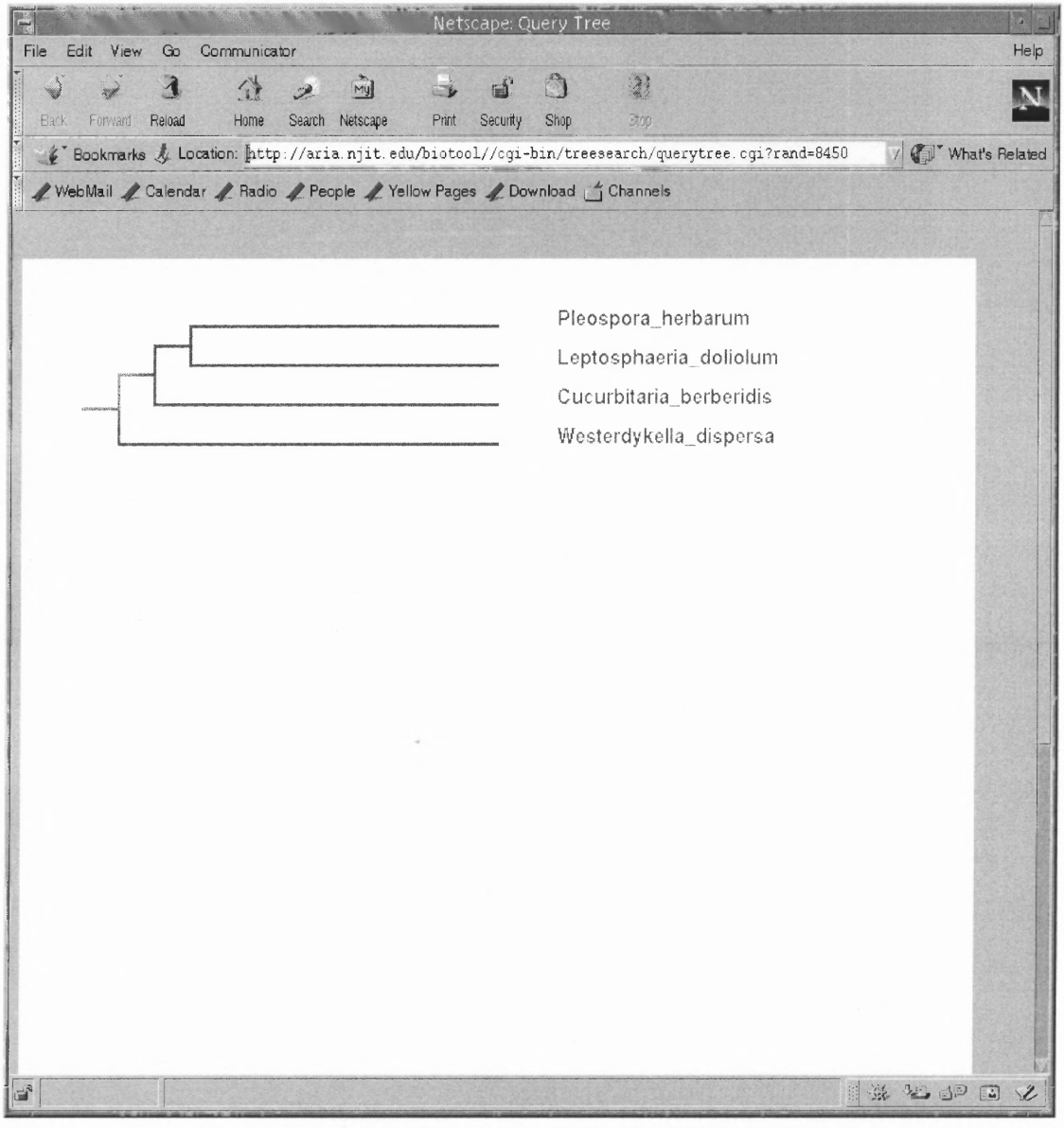


Figure 5.2 Query tree display.

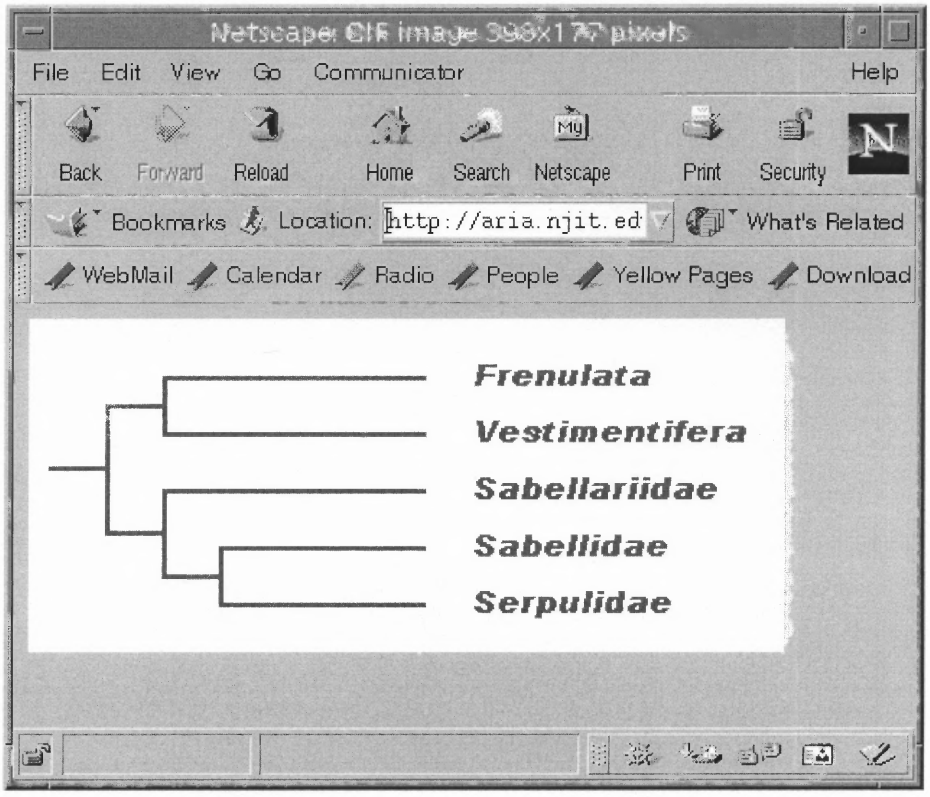


Figure 5.3 An example query tree.

5.2 Input

There are two ways to submit the query:

(A) type input query tree to the text area, then press the "Submit" button.

(B) choose the query tree file, then press the "Upload" button.

The followings are valid input format samples:

The valid input format for the query tree in Figure 5.3 is ((Frenulata,Vesti mentifera), (Sabellariidae,(Sabellidae,Serpulidae))). The valid input format for a query tree with only one node: the taxon name, such as "Galidia elegans".

5.3 View Query Tree and Search Results

The query tree and matching trees can be viewed graphically or in text format. Users can click the [Text] to view the text representation of a tree. By clicking the tree icon, users can view the graphic representation of a tree. The matching nodes are shown in bold font and with a green dot. This search engine can guide the users to navigate the neighboring trees of query tree as shown in Figure 5.5.

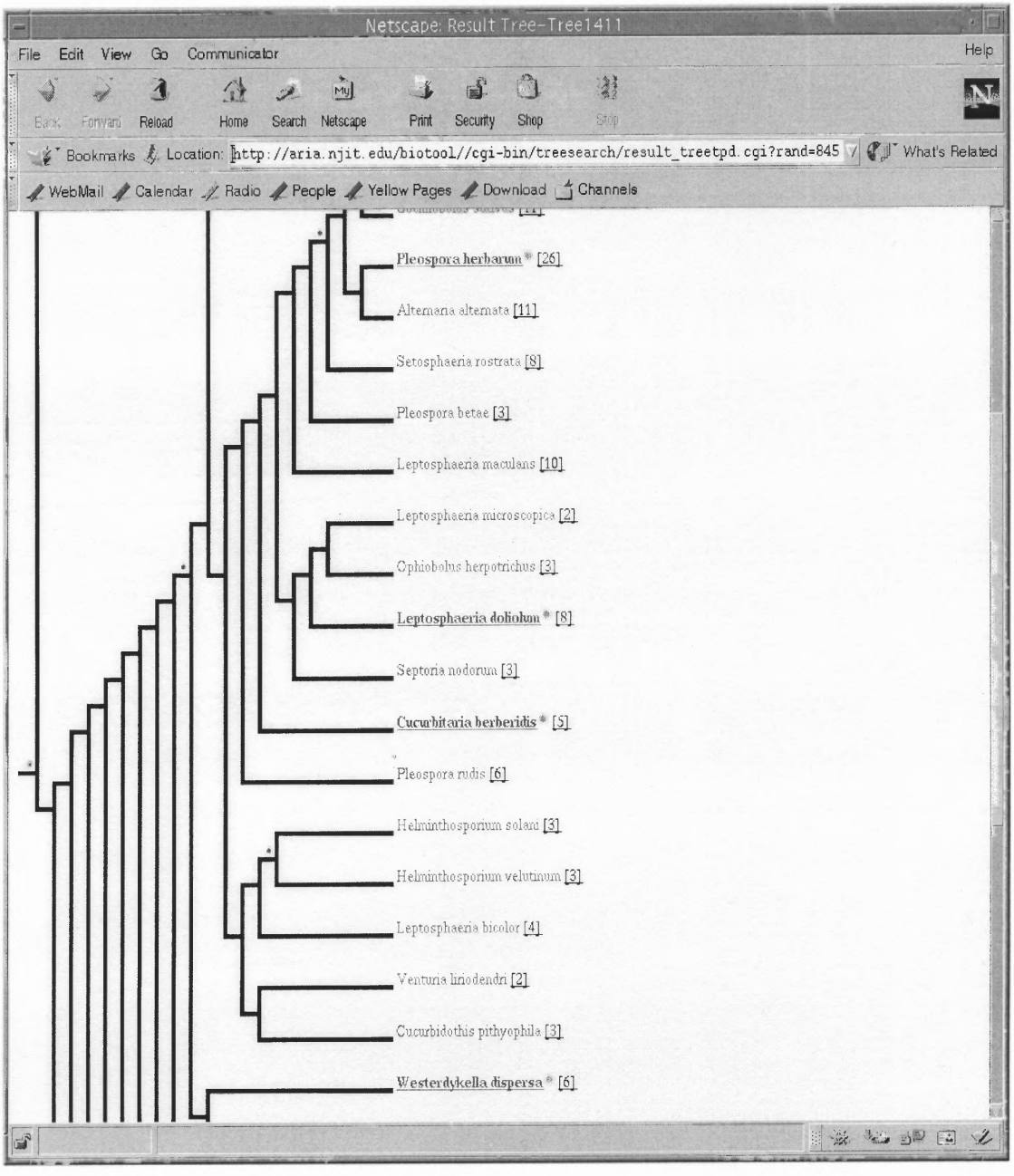


Figure 5.4 Matching data tree display.

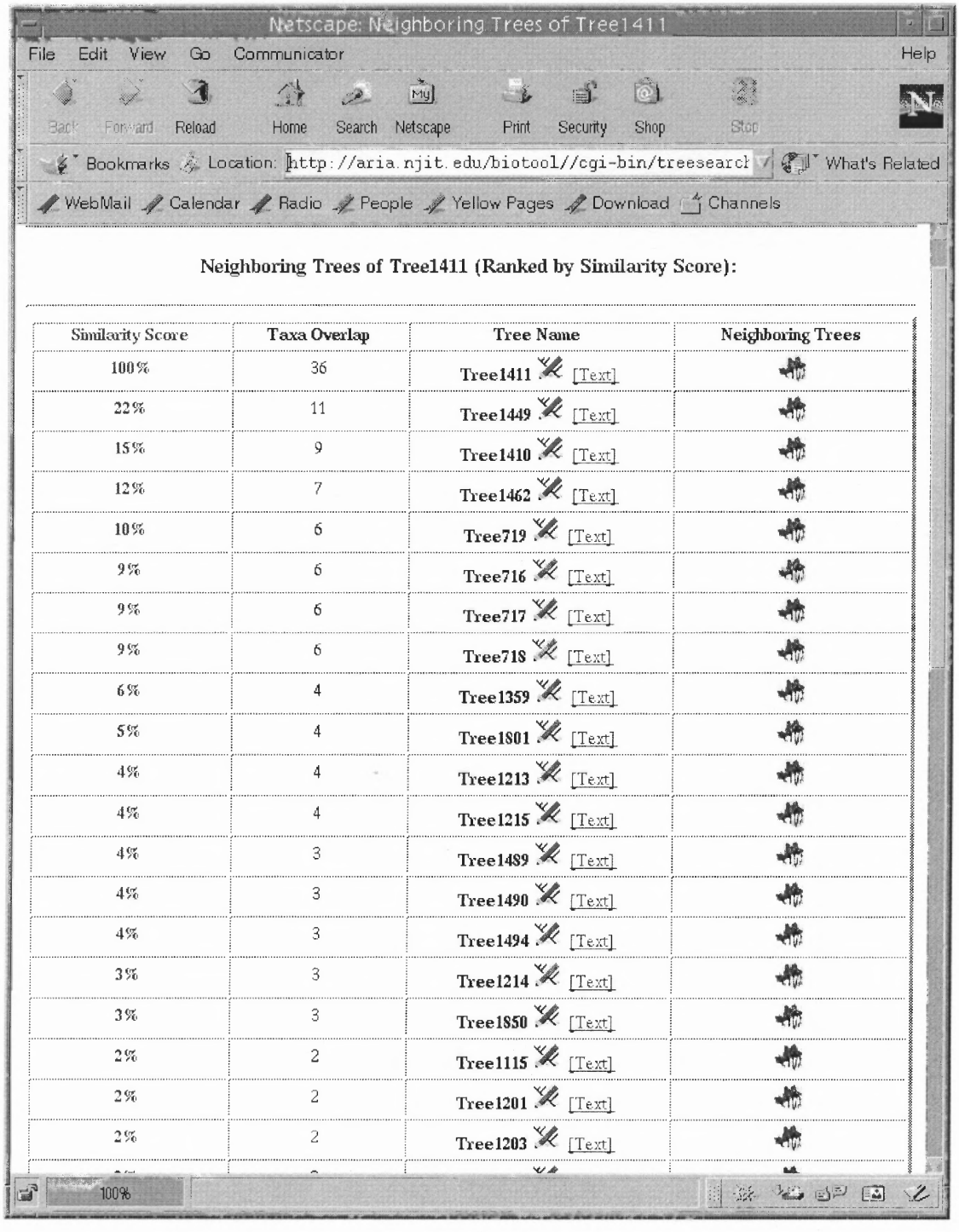


Figure 5.5 Neighboring trees display.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

Labeled trees find many applications in computer and natural sciences. a new approach, called ATreeGrep, has been presented for searching among these trees. Experimental results show that ATreeGrep is fast, particularly when the dictionary size of node labels is large. This algorithm has been implemented into two Web-based search engines for phylogenetic databases and XML repositories.

A new approach to nearest neighbor searching among phylogenetic trees is also presented. Given a query or pattern tree P and a database of trees $\mathcal{D}$, the proposed approach finds and ranks data trees D where D is similar to P, and D shares at least one common leaf label with P. If D doesn't share any common leaf label with P, D will be eliminated without computing the similarity score from P to D.

In practice, another interesting search function is to find data trees in $\mathcal{D}$ that are within UpDown distance ϵ of the query tree P. A filter has been implemented to speed up this type of retrieval, which works as follows. For the database of trees, a hash table keyed by pair of node labels and each hash bin contains tree identification numbers is created. The pair can be in alphabetical order because $U[u, v] = D[v, u]$ for any pair of node labels (u, v) (cf. Fact 1 in Section 3). Now given the query tree P, each pair of node labels in P is considered to see which trees of the database the pair is in. (This requires time independent of the size of the database). Sort the data trees by the number of hits.

By evaluating a data tree D, a lower bound on the distance from P to D can be got by looking at $U_P[u, v]$, where U_P is the UpDown matrix of P and (u, v) is a pair in P that is missing from D. The lower bound is computed by summing up $U_P[u, v]$ for all pairs of (u, v) of P that are missing from D. If the lower bound is already greater

than ϵ, D can be eliminated from consideration without calculating the distance of P and D. Furthermore, if a data tree D has a set S of k hits and it is decided D doesn't qualify to be a solution after calculating the distance of P and D, then any data tree D' that only has S' or less than k hits where S' is a subset of S will not be a solution and hence can be eliminated from consideration. Experimentally this filter helps to eliminate, on average, more than 95% of data trees while performing nearest neighbor searches in TreeBASE.

The proposed algorithms have been used for performing structure-based retrieval and for analyzing the structures of phylogenetic trees in TreeBASE. Future work includes extending the discussed algorithms for searching other types of scientific databases, and for finding patterns in these databases [61, 63].

APPENDIX

PARTIAL PROGRAM LISTING

The proposed algorithms and the Web-based systems are implemented using Java, Perl and C. This appendix includes the partial source code for the structural search engine.

A.1 nettreedraw.java

```
/*************************************************************************/
/*      Program: nettreedraw.java                                        */
/*      (c) Copyright 2004                                               */
/*      All rights reserved                                              */
/*      Programs written by Huiyuan Shan (NJIT)                          */
/*      RA in the group of Jason T. L. Wang (New Jersey Institute        */
/*      of Technology) and Dennis Shasha (New York University)           */
/*                                                                       */
/*      Permission to use, copy, modify, and distribute this             */
/*      software and its documentation for any purpose and without       */
/*      fee is hereby granted, provided that this copyright              */
/*      notice appears in all copies.   Programmer(s) makes no           */
/*      representations about the suitability of this                    */
/*      software for any purpose.  It is provided "as is" without        */
/*      express or implied warranty.                                     */
/*                                                                       */
/*************************************************************************/
import java.io.*;
import java.util.*;
import java.math.*;
import java.awt.*;
import java.applet.*;
import java.net.*;

public class nettreedraw {
   static Vector v_all   =new Vector(200,1);
   static int count_elm=0;

   public static void main(String[] args) throws Exception {
     if(args.length>=1)
```

```
{
  StringBuffer sb=null;
  StringBuffer sb2=null;
  boolean  TitleBegin=false;
  boolean firsttime=true;
  StringBuffer buf = new StringBuffer();
  BufferedReader fin = null;
  String line;

  sb=new StringBuffer(0);
  sb=sb.append(args[0]);
  sb2=new StringBuffer(0);
  sb2=sb2.append(args[1]);

  try {
     fin = new BufferedReader(new FileReader(sb2.toString()));
     while ((line = fin.readLine()) != null)
       {
         int nbsp=line.indexOf(" ");
         while(nbsp>=0){
         count_elm++;
         v_all.addElement((line.substring(0,nbsp)
            ).replace('_',' ').trim());
         line=line.substring(nbsp+1);
         nbsp=line.indexOf(" ");
       }
    }

 }
 catch (Exception e) {
     System.out.println("Error: " + e.toString());
   }
 v_all.trimToSize();

 URL treebase = new URL("http://128.205.184.102/treebase
       /TreeBASE.acgi?PickedItems="+sb.toString()
       +"&PickedItems=1&Button=Tree");
 URLConnection yc = treebase.openConnection();
 BufferedReader in = new BufferedReader(
                  new InputStreamReader(
                  yc.getInputStream()));
 String inputLine;

 while ((inputLine = in.readLine()) != null)
        System.out.println(addTags(inputLine));
```

```
        }
       else System.out.println("Invalid Argument");
    }

   public static String addTags(String s)
   {
   boolean CladFound=false;
   String s2=null;
   int nbsp=s.indexOf("<FONT SIZE=3 COLOR=\"#");
     if(nbsp>=0){
      int nbsp2=s.indexOf("</FONT>");
      if(nbsp2>=0){
        s2=new String(s.substring(nbsp+30,nbsp2));
        for(int i=0; i<v_all.size(); i++)
          { if(v_all.elementAt(i).toString().equals(s2))
                 {
                 CladFound=true;
                 break;  }
          }
        if(CladFound) s=s.substring(0,nbsp+30)+"<U><b>"+s2+"</b></U>"
           +s.substring(nbsp2,nbsp2+7) +
           "<IMG SRC=\"match.jpg\" BORDER=0>"+
            s.substring(nbsp2+7);
        return s;
      }
     else return s;
    }
    else return s;
  }
}
```

A.2 querytree.java

```
/*****************************************************************/
/*      Program: querytree.java                                  */
/*      (c) Copyright 2004                                       */
/*      All rights reserved                                      */
/*      Programs written by Huiyuan Shan (NJIT)                  */
/*      RA in the group of Jason T. L. Wang (New Jersey Institute */
/*      of Technology) and Dennis Shasha (New York University)   */
/*                                                               */
/*      Permission to use, copy, modify, and distribute this     */
/*      software and its documentation for any purpose and without */
/*      fee is hereby granted, provided that this copyright      */
/*      notice appears in all copies.   Programmer(s) makes no   */
/*      representations about the suitability of this            */
/*      software for any purpose.  It is provided "as is" without */
/*      express or implied warranty.                             */
/*                                                               */
/*****************************************************************/
import java.io.*;
import java.util.*;
import java.math.*;
import java.awt.*;
import java.applet.*;
import java.net.*;

/**
 * This class reads PARAM tags from its HTML host page and sets
 * the color and label properties of the applet. Program execution
 * begins with the init() method.
 */
public class querytree extends Applet
{
  public void init()
  {
     initForm();
     usePageParams();
   }

  private final String labelParam = "label";
  private final String backgroundParam = "background";
  private final String foregroundParam = "foreground";
  private String  FileURL=null;

  private void usePageParams()
```

```
{
 final String defaultLabel = "Default label";
 final String defaultBackground = "C0C0C0";
 final String defaultForeground = "000000";
 String labelValue;
 String backgroundValue;
 String foregroundValue;

 labelValue = getParameter(labelParam);
 backgroundValue = getParameter(backgroundParam);
 foregroundValue = getParameter(foregroundParam);

 FileURL=getParameter("parseStringFromApplet");
 if ((labelValue == null) || (backgroundValue == null) ||
                      (foregroundValue == null))
 {

  labelValue = defaultLabel;
  backgroundValue = defaultBackground;
  foregroundValue = defaultForeground;
  }

 label1.setText(labelValue);
 label1.setBackground(stringToColor(backgroundValue));
 label1.setForeground(stringToColor(foregroundValue));
 this.setBackground(stringToColor(backgroundValue));
 this.setForeground(stringToColor(foregroundValue));
 }

private Color stringToColor(String paramValue)
{
  int red;
  int green;
  int blue;

  red = (Integer.decode("0x" + paramValue.substring(0,2)))
         .intValue();
  green = (Integer.decode("0x" + paramValue.substring(2,4)))
         .intValue();
  blue = (Integer.decode("0x" + paramValue.substring(4,6)))
         .intValue();
  return new Color(red,green,blue);
}

public String[][] getParameterInfo()
```

```
{
 String[][] info =
 {
  { labelParam, "String", "Label string to be displayed" },
  { backgroundParam, "String", "Background color,
    format \"rrggbb\"" },
  { foregroundParam, "String", "Foreground color,
    format \"rrggbb\"" },
              };
  return info;
 }

Label label1 = new Label();

void initForm()
{
  this.setBackground(Color.lightGray);
  this.setForeground(Color.black);
  label1.setText("label1");
  this.setLayout(new BorderLayout());
  this.add("North",label1);
}

public void paint(Graphics g)
{
 try {
   URL theURL = new URL(FileURL);
   BufferedReader input = new BufferedReader(new
   InputStreamReader(theURL.openStream()));
   g.setFont(new Font("Arial Black", Font.BOLD, 16));
   String parseString = input.readLine();
   input.close();

   int OX  = 450;
   int OX_end = 450-50;
   int OX_head =50;
   int OY  = 50;
   int OW  = 30;
   int OH  = 30;
   int highest = 0;
   String st_temp;
   int k = 0;

   StringBuffer sb = new StringBuffer(parseString);
   int j=0;
```

```
while ( j<sb.length())
 {
    char ch = sb.charAt(j);
    if(ch=='(') sb.insert(j+1,"#");
    else if(ch==')')
       sb.insert(j++,"#");
         else if(ch==',')
                {
                  sb.insert(j+1,"#");
                  sb.insert(j++,"#");
                 }
         j++;
 }

 Vector v_name   = new Vector(100,1);
 Vector v_all    = new Vector(500,1);
 Vector v_ad     = new Vector(500,1);
 StringTokenizer st = new StringTokenizer(sb.toString(), "#");

 while (st.hasMoreTokens())
 {
   st_temp = st.nextToken();
   v_all.addElement(st_temp);
   if(st_temp.equals("(") || st_temp.equals(")")
     || st_temp.equals(","))
     v_ad.addElement(st_temp);
   else if(!st_temp.equals(","))
            v_ad.addElement("#");
   if(!st_temp.equals(")") && st_temp.charAt(0)==')
      v_ad.addElement("~"+st_temp);
   if( st_temp.charAt(0)!=')' &&!st_temp.equals("(")
     && !st_temp.equals(")") && !st_temp.equals(",") )
     v_name.addElement(st_temp);
 }

 v_all.trimToSize();
 v_name.trimToSize();
 v_ad.trimToSize();

 int [] v_name_depth  = new int[100];
 int current_depth = 0;
 j = 0;
 int [] X_position = new int[100];
 int [] Y_position = new int[100];
```

```
for(int i=0; i<v_all.size(); i++)
{
  st_temp = v_all.elementAt(i).toString();
  if(st_temp.equals("("))
      current_depth++;
  if(st_temp.equals(")"))
      current_depth--;
  else if(st_temp.charAt(0)==')')
       current_depth--;
  if ( st_temp.charAt(0)!=')' && !st_temp.equals("(")
     && !st_temp.equals(")") && !st_temp.equals(",") )
      v_name_depth[j++] = current_depth;
  if(current_depth > highest)  highest=current_depth;
 }

int max_n = j--;

for(int i=0; i<v_name.size(); i++)
{
  X_position[i] = OX_head+v_name_depth[i]*OW;
  Y_position[i] = OY+i*OH;

  g.drawString(v_name.elementAt(i).toString(),OX,50+i*OH);
  g.drawLine(OX_end, 50+i*OH,
  X_position[i], 50+i*OH);
  g.drawLine(OX_end, 50+i*OH+1,
  X_position[i], 50+i*OH+1);
  g.drawLine(OX_end, 50+i*OH+2,
  X_position[i], 50+i*OH+2);
  g.setColor(Color.red);
 }

for(int h=highest; h>0; h--)
{
  for(int a=0, n=0; a<v_ad.size(); a++)
  {
    int item = 1;
    if(v_ad.elementAt(a).toString().equals("(") &&
       v_ad.elementAt(a+1).toString().equals("#") &&
       v_name_depth[n]==h)
     {
      v_ad.removeElementAt(a);
      v_ad.removeElementAt(a);
      int head_x = 0;
```

```
    int head_y = 0;
    if(v_name_depth[n]==h)
    {
      head_x = X_position[n];
      head_y = Y_position[n];
    }
    v_name_depth[n] = v_name_depth[n]-1;
    int end_x=0;
    int end_y=0;

    while((!v_ad.elementAt(a).toString().equals(")"))
    && (v_ad.elementAt(a).toString().charAt(0)!='~') )
    {
       if(v_ad.elementAt(a).toString().equals("#"))
        item++;
       v_ad.removeElementAt(a);
     }

    if  (v_ad.elementAt(a).toString().charAt(0)=='~') item--;
    end_x=X_position[n+item-1];
    end_y=Y_position[n+item-1];
    g.drawLine(head_x,head_y,end_x,end_y);
    g.drawLine(head_x+1,head_y,end_x+1,end_y);
    g.drawLine(head_x+2,head_y,end_x+2,end_y);
    g.drawLine(head_x,(head_y+end_y)/2,end_x-OW,
               (head_y+end_y)/2);
    g.drawLine(head_x,(head_y+end_y)/2+1,end_x-OW,
               (head_y+end_y)/2+1);
    g.drawLine(head_x,(head_y+end_y)/2+2,end_x-OW,
               (head_y+end_y)/2+2);
    if  (v_ad.elementAt(a).toString().charAt(0)=='~')
       g.drawString(v_ad.elementAt(a).toString().substring(2),
                    head_x-15,(head_y+end_y)/2);
    v_ad.setElementAt("#",a);
    X_position[n]=end_x-OW;
    Y_position[n]=(head_y+end_y)/2;

    for(int temp_n=n+1; temp_n<max_n; temp_n++)
    {
      Y_position[temp_n] = Y_position[temp_n+item-1];
      X_position[temp_n] = X_position[temp_n+item-1];
      v_name_depth[temp_n] = v_name_depth[temp_n+item-1];
     }
  max_n = max_n-item+1;
  n = n+1;
```

```
          }
          else if(v_ad.elementAt(a).toString().equals("#"))
            n++;
          v_ad.trimToSize();
         }
        String displayString = "";
      }
    }
    catch(Exception e)     {}
}
```

A.3 Transform.pl

```
%/*****************************************************************/
%/*     Program: Transform.pl                                     */
%/*     (c) Copyright 2004                                        */
%/*     All rights reserved                                       */
%/*     Programs written by Huiyuan Shan (NJIT)                   */
%/*     RA in the group of Jason T. L. Wang (New Jersey Institute */
%/*     of Technology) and Dennis Shasha (New York University)    */
%/*                                                               */
%/*     Permission to use, copy, modify, and distribute this      */
%/*     software and its documentation for any purpose and without */
%/*     fee is hereby granted, provided that this copyright       */
%/*     notice appears in all copies.   Programmer(s) makes no    */
%/*     representations about the suitability of this             */
%/*     software for any purpose.  It is provided "as is" without */
%/*     express or implied warranty.                              */
%/*                                                               */
%/*****************************************************************/
#! /usr/local/bin/perl
# 0--undefined,1--TreeBase, 2--MacClade

$dir1 = "/home/eservice/cltools/data/tree/";
$input = $ARGV[0];
$class= $ARGV[1];

open(FH,"< $input")|| die "cannot open $input";
$flag=0;
my $testpattern="TREE.+,.+;";
$count=0;


while ($line=<FH>){
 if($class==0) {
   if ($line=~/^\[File.+TreeBASE./) { $class=1; }
   if ($line=~/\[MacClade.+HARVARD/) { $class=2; }
  }
 if($class==1) {
       if ($line=~/[0-9]\t/){
          $line =~s/\'//g;
          @b=split(/\t\t/,$line);
          @a=split(/\t/,$b[1]);
          $num=$a[0];
          $a[1]=~s/,//;
          $a[1]=~s/\'//g;
```

```
        $label=$a[1];
        $numlabel{$num}=$label;
        print "$num"."="."$numlabel{$num}";
       }
    if ($line=~/^\[!.+#:.+\]/){
        print "$line";
        @n1=split(/#:\s+/,$line);
        @n2=split(/\s/,$n1[1]);
        print "$n1[1]";
        print "\n";
     }
    if ($line=~/$testpattern/){
        $dest="$dir1"."$n2[$count]";
        open(FH1,">$dest")||die "cannot open temp";
        @node=split(/R\]\s/,$line);
        foreach $num(keys %numlabel){
            print "$num"."="."$numlabel{$num}";
            $node[1]=~s/,$num/,$numlabel{$num}/g;
            $node[1]=~s/\($num/\($numlabel{$num}/g;
            #print ",$num\n";
           }
        $node[1]=~s/\s//g;$node[1]=~s/\;//;
        print "$node[1]\n";
        print FH1  "$node[1]";
        close(FH1);
        $count++;
    }
}

if($class==2) {
    if ($line=~/[0-9]\t/){
        $line =~s/\'//g;
        @b=split(/\t\t/,$line);
        @a=split(/\t/,$b[1]);
        $num=$a[0];
        $a[1]=~s/,//;
        $a[1]=~s/\'//g;
        $label=$a[1];
        $numlabel{$num}=$label;
        print "$num"."="."$numlabel{$num}";
       }
    if ($line=~/^\[!.+#:.+\]/){
        print "$line";
        @n1=split(/#:\s+/,$line);
        @n2=split(/\]/,$n1[1]);
```

```
            $n2[0]=~s/\//x/g;
           print "$n2[0]";
           $n2[0]=~s/:/c/g;
           print "$n2[0]";
           print "\n";
         }
        if ($line=~/$testpattern/){
           $dest="$dir1"."$n2[0]$count";
           open(FH1,">$dest")||die "cannot open temp";
           @node=split(/R\]\s/,$line);
        foreach $num(keys %numlabel){
               print "$num"."="."$numlabel{$num}";
               $node[1]=~s/,$num/,$numlabel{$num}/g;
               $node[1]=~s/\($num/\($numlabel{$num}/g;
         }
        $node[1]=~s/\s//g;$node[1]=~s/\;//;
        print "$node[1]\n";
        print FH1  "$node[1]";
        close(FH1);
        $count++;
        }
    }
}
close(FH);
```

A.4 Partition.pl

```
%/*****************************************************************/
%/*     Program: Partition.pl                                     */
%/*     (c) Copyright 2004                                        */
%/*     All rights reserved                                       */
%/*     Programs written by Huiyuan Shan (NJIT)                   */
%/*     RA in the group of Jason T. L. Wang (New Jersey Institute */
%/*     of Technology) and Dennis Shasha (New York University)    */
%/*                                                               */
%/*     Permission to use, copy, modify, and distribute this      */
%/*     software and its documentation for any purpose and without */
%/*     fee is hereby granted, provided that this copyright       */
%/*     notice appears in all copies.   Programmer(s) makes no    */
%/*     representations about the suitability of this             */
%/*     software for any purpose.  It is provided "as is" without */
%/*     express or implied warranty.                              */
%/*                                                               */
%/*****************************************************************/
#!/usr/local/bin/perl
 @proglist = ("echo", "Please Wait .... Transform in progress!");
 system(@proglist);

 $Partition=$ARGV[2];
 $threshold=1000/$Partition+0.001;
 $ct=$threshold;

 $dirres="/home/eservice/cltools/";
 $makepathfix = "/home/biosoft/cltools/trans";
 $res="/home/biosoft/cltools/res";
 $resbak="/home/biosoft/cltools/resbak";
 $label="/home/biosoft/cltools/treelabel";
 $labelbak="/home/biosoft/cltools/treelabelbak";
 $count=0;
 $part=1;
 $partnum=0;

 system("cp $resbak $res");
 system("cp $labelbak $label");

 $input=$ARGV[0];
 $target=$ARGV[1];

 open(FH,"< $input")|| die "cannot open $input";
```

```
 while ($line=<FH>){
  $count++;
  $partnum++;
  print "$count\n";
  @b=split(/\t/,$line);
  open(FH1,">temp")||die "cannot open temp";
  print FH1  "$b[1]";
  close(FH1);
  system("$makepathfix temp $b[0]");
  if ($partnum>=$threshold) {
    system("cat $label>>$res ");
    system("rm $label");
    system("mv $res $target$part");
    $part++;
    $threshold= $threshold+$ct;
    system("cp $resbak $res");
    system("cp $labelbak $label");
   }
 }

 print "$count\n";
 if ($partnum < $threshold) {
   system("cat $label>>$res ");
   system("rm $label");
   system("mv $res $target$part");
 }
closedir(FH);
```

A.5 Updown.C

```
/*****************************************************************/
/*      Program: Updown.C                                        */
/*      (c) Copyright 2004                                       */
/*      All rights reserved                                      */
/*      Programs written by Huiyuan Shan (NJIT)                  */
/*      RA in the group of Jason T. L. Wang (New Jersey Institute */
/*      of Technology) and Dennis Shasha (New York University)   */
/*                                                               */
/*      Permission to use, copy, modify, and distribute this     */
/*      software and its documentation for any purpose and without */
/*      fee is hereby granted, provided that this copyright      */
/*      notice appears in all copies.   Programmer(s) makes no   */
/*      representations about the suitability of this            */
/*      software for any purpose.  It is provided "as is" without */
/*      express or implied warranty.                             */
/*                                                               */
/*****************************************************************/

#include <stdio.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>

#define MAXNODE 400

struct Tree {
    int vertex_no;
    char info[256];
    int numchild;
    int tag_marked;
    int c_id;
    struct Tree *parent;
    struct Tree *left_child;
    struct Tree *next_sibling;
};

FILE *fp_h;
FILE *fp_res;

struct Tree *T1,*T2;
struct Tree TB[MAXNODE*2];
char node_info[256];
char treeid[256];
```

```
char pl1[MAXNODE][256],pl2[MAXNODE][256];
char ndl1[MAXNODE][256],ndl2[MAXNODE][256];
static int m1u[MAXNODE][MAXNODE],m2u[MAXNODE][MAXNODE], m1d[MAXNODE]
[MAXNODE],m2d[MAXNODE][MAXNODE];
int TB_counter=0,TB_q;
int depth=0;
int vertexno=0;
int  subtree_num_nodes;
int  table[MAXNODE][2];
int  table_index[MAXNODE];
int meaningful=0;
int number_matched=0;
int nodenum1,nodenum2;
int tableq[MAXNODE][2];
int re_count=1;
int relabel_count=1;
float relative_ranking;

struct Tree * mallocT();
void calculate_matrix(struct Tree *root,int u[MAXNODE][MAXNODE],int d
[MAXNODE][MAXNODE],int num );
void matrix_print(int  p[MAXNODE][MAXNODE],int n);
void cal_table();
int  compare_matrix();
char get_next_char();
int get_next_token(char * str,int mode);
void construct_tree_preord(struct  Tree *T);
void construct_tree_prord(struct  Tree *T);
void construct_tree_pstord(struct  Tree *T);
int check_order(char *fn);
int  to_pd_tree(struct Tree *T);
int  to_pd_label(struct Tree *T,char ndl[MAXNODE][256]);
struct Tree * reduce_rd_nodes(struct Tree *T);
struct Tree * merge_neighbor(struct Tree *T);
struct Tree * mark_one(struct Tree *T);
void  to_Harvard(struct Tree *T);
void  relabel(struct Tree *T);
void gc(struct Tree *T);
void preprocess(struct Tree *T);
void mark_labelq(struct Tree *T);
int refine_mark(struct Tree *T);
void shrink(struct Tree *T);


int main(int argc,char *argv[])
```

```
{
 struct Tree  *tp;
 char fn1[256], fn2[256];
 char T_s[20], T_t[20];
 char pch='%';
 int  tree_order;
 int count_id=2;
 int dit_int;


 if (argc!=3) {
     printf("Invalid argument  \n");
     exit(-1);
    };

  strcpy(fn1,argv[1]);
  strcpy(fn2,argv[2]);
  strcpy(treeid,"Tree1");

  tree_order=check_order(fn1);

  if (tree_order==-1) {
     printf("\nInvalid Tree Representation \n");
     return(-1);
     }

  tree_order=1;
  check_order(fn2);

  if (tree_order==-1) {
     printf("\nInvalid Tree Representation \n");
     return(-1);
     }

 if ((fp_h=fopen(fn1,"r"))==NULL)
    {  printf( "\n Can not open file %s",fn1);
       exit(0);
       }

 if ((T1=(struct Tree *)mallocT())==NULL)
     {  printf( "Allocation memory error \n");
        exit(0);
     }
  if (tree_order==1) {
       construct_tree_pstord(T1);
```

```
  }
else  construct_tree_pstord(T1);

fclose(fp_h);
tp=T1;
T1=T1->left_child;
T1->parent=NULL;

to_pd_tree(T1);
preprocess(T1);
to_pd_label(T1,ndl1);

nodenum1 =vertexno-1;
depth=0;
vertexno=0;
get_next_token(node_info,0);
strcpy(treeid,"Tree2");

if ((fp_h=fopen(fn2,"r"))==NULL)
  {  printf( "\n Can not open file %s",fn2);
     exit(0);
     }

TB_q=TB_counter+1;

while(!feof(fp_h)){

TB_counter=TB_q;
meaningful=0;
number_matched=0;
depth=0;
vertexno=0;
re_count=1;
fscanf(fp_h,"%s",treeid);
get_next_token(node_info,0);
strcpy(T_s,"Tree");
sprintf(T_t,"%d",count_id++);

if ((T2=(struct Tree *)mallocT())==NULL) {
     printf( "Allocation memory error \n");
     exit(0);
   }
construct_tree_pstord(T2);
nodenum2 =vertexno-1;
if(nodenum2<=1) {
```

```
        count_id--;
        continue;
    }

    tp=T2;
    T2=T2->left_child;
    T2->parent=NULL;
    to_pd_tree(T2);

    mark_labelq(T1);

    refine_mark(T1);
    refine_mark(T2);
    T2= reduce_rd_nodes(T2);

    if(T2==NULL) {
     continue;
    }
    to_Harvard(T2);
    printf("*");

    T2=merge_neighbor(T2);
    cal_table();
    relabel_count=1;
    relabel(T2);

    calculate_matrix(T2,m2u,m2d,nodenum2);
    matrix_print(m2u,nodenum2);
    calculate_matrix(T1,m1u,m1d,nodenum1 );
    matrix_print(m1u,nodenum1);
    dit_int=compare_matrix();

    if( relative_ranking>=0.0) relative_ranking=sqrt(relative_ranking);
      else if (relative_ranking >-0.001)
             relative_ranking=sqrt(- relative_ranking);
           else  relative_ranking=-sqrt(-relative_ranking);
    printf("\n%2.0f%c-",relative_ranking*100,pch);
    printf(" %s",treeid);
    printf("- %d", number_matched);

    }
}


struct Tree * mallocT()
```

```
{
return (&TB[TB_counter++]);
}

int  compare_matrix()
{
 int i,j;
 int all_dist=0;
 int total_dist=0;

 for (i=0;i<meaningful;i++)
   for (j=0;j<meaningful;j++)
     all_dist+=m1u[table[i][0]][table[j][0]];

 for (i=0;i<meaningful;i++)
   for (j=0;j<meaningful;j++)
     {
      if ((table[i][1]==-MAXNODE)||(table[j][1]==-MAXNODE))
          total_dist+=abs(m1u[table[i][0]][table[j][0]]);
       else
          total_dist+=abs(m1u[table[i][0]][table[j][0]]
                  -m2u[table[i][1]][table[j][1]]) ;
     }

  relative_ranking= 1.0- ((float)  total_dist)/all_dist;
  ALLD=all_dist;
  return total_dist;
}

void cal_table()
{
 int i,j;

 for (i=0;i<nodenum1;i++)
  {
    if ((strlen( ndl1[i])!=0)&&(tableq[i][1]==1)) {
      for(j=0;j<nodenum2;j++)
        if (!strcmp(ndl1[i],ndl2[j])) {
             table[meaningful][0]=i;
             table[meaningful][1]=j;
             number_matched++;
             meaningful++;
             break;
        }
```

```
if(j==nodenum2) {
              table[meaningful][0]=i;
              table[meaningful][1]=-MAXNODE;
              meaningful++;
          }
        }
    }
}


void calculate_matrix(struct Tree *T,int u[MAXNODE][MAXNODE],int d
[MAXNODE][MAXNODE],int num )
{

 struct Tree *temp;
 int i,c,parentid;

  if (T!=NULL) {
     if(T->parent!=NULL){
       parentid=T->parent->vertex_no;
       c=T->vertex_no;
       u[c-1][c-1]=0;
       d[c-1][c-1]=0;
       u[parentid-1][c-1]=0;
       if(T->parent->tag_marked==1) {
           u[c-1][parentid-1]=1;
           d[parentid-1][c-1]=1;
        }
       else
        { u[c-1][parentid-1]=0;
          d[parentid-1][c-1]=0;
        }
       d[c-1][parentid-1]=0;

       for (i=1;i<c;i++)
         { if ((i!=parentid)&&(i!=c))
             {
                u[i-1][c-1]=  u[i-1][parentid-1];
                d[c-1][i-1]= d[parentid-1][i-1] ;
                if(T->parent->tag_marked==1) {
                 u[c-1][i-1]= 1+u[parentid-1][i-1];
                 d[i-1][c-1]= 1+d[i-1][parentid-1];
                }
                else {
                 u[c-1][i-1]= u[parentid-1][i-1];
```

```
                d[i-1][c-1]= d[i-1][parentid-1];
               }
             }
           }
         }
       }
   if(T!=NULL) {
      if ((temp=T->left_child)!=NULL) {
       while (temp!=NULL) {
       calculate_matrix(temp,u,d,num);
        temp=temp->next_sibling;
        }
      }
    }
}


/* Garbage Collection */

void gc(struct Tree *T)
{
 struct Tree *temp1=NULL,*temp2=NULL;

  if (T!=NULL) {
      if ((temp1=T->left_child)!=NULL) {
         while (temp1!=NULL) {
         temp2=temp1;
         temp1=temp1->next_sibling;
         gc(temp2);
        }
       }
      free(T);
    }
}


void preprocess(struct Tree *T)
{

 struct Tree *temp;

  if (T!=NULL) {
     if(T->left_child !=NULL) tableq[T->vertex_no-1][0]=0;
  else tableq[T->vertex_no-1][0]=1;
     }
```

```
 if ((temp=T->left_child)!=NULL) {
       while (temp!=NULL) {
        preprocess(temp);
        temp=temp->next_sibling;
        }
   }

}


int refine_mark(struct Tree *T)
{
  struct Tree *temp;
  int i=0;

  if (T!=NULL) {

  if ( (temp=T->left_child)==NULL) return T->tag_marked;
   else{
       while (temp!=NULL) {
        i+=refine_mark(temp);
        temp=temp->next_sibling;
        }
           if (i>1) { if (T->tag_marked==0)
                      T->c_id=-99;
              T->tag_marked=1 ;
              T->numchild=i;
              return 1;
           }
           else {
                 T->numchild=i;
                 return i;
           }
      }
    }
 }


struct Tree * reduce_rd_nodes(struct Tree *T)
{
  struct Tree *temp,*tp2,*tp3,*tpd,*pre,*res,*final;
  int i=0;

  tp3=NULL;
```

```
 tp2=NULL;
 final=NULL;

 if (T!=NULL) {

    if  (T->tag_marked==0) {
      final=NULL;
      temp=T->left_child;
      while (temp!=NULL) {
        tpd=temp;
        tp3=temp->next_sibling;
        res=reduce_rd_nodes(temp);
        temp=tp3;
        if (res!=NULL) final=res;

        }
       if (final!=NULL) {
          final->parent=NULL;
          final->next_sibling=NULL;
         }
       return final;
     }
    else {
      temp=T->left_child;
      T->left_child=NULL;
      tp2=NULL;
      while (temp!=NULL) {
         res=reduce_rd_nodes(temp);
         tp3=temp;
         temp=temp->next_sibling;
         if (res!=NULL) {
           if (T->left_child==NULL) T->left_child =res;
             else tp2->next_sibling=res;
           res->parent=T;
           tp2=res;

           }
        }
        if(tp2!=NULL) tp2->next_sibling=NULL;
        return T;
      }
 }
 else return NULL;
}
```

```
struct Tree *  merge(  struct Tree *T)
{
   struct Tree *temp,*parent,*tp2,*tp3,*pre,*parent_choosen;
   int i=0;
   int id=-99, final=1,mergetag,par_id,tbok=1,ppid;

   pre=NULL;
   parent=NULL;
   temp=T->parent->left_child;
   while ((temp!=NULL)&&(temp!=T)) {
    if (temp->c_id==T->c_id) {
    parent=temp;
    break;
    }
    pre=temp;
    temp=temp->next_sibling;
   }

   if ((parent!=NULL)&&(parent!=T)) { /*Merge */
     temp=T->parent->left_child;
     while ((temp!=NULL)&&(temp!=T)) {
        pre=temp;
        temp=temp->next_sibling;
     }

   pre->next_sibling=T->next_sibling;
   temp=T->left_child;
   while (temp!=NULL) {
      tp2=temp;
      temp=temp->next_sibling;
    }
   tp2->next_sibling=parent->left_child;
   parent->left_child=T->left_child;
   tp2=T->parent;
   free(T);
  }
  else {
    parent=T;
    tp2=T->parent;
  }

 if(tp2->c_id<0) {
   mergetag=1;
   ppid=-99;
   temp= parent->parent->left_child;
```

```
 parent_choosen=NULL;
 while (temp!=NULL) {
   if ((temp->left_child!=NULL)&&(temp!=parent))
      {  if(ppid=-99) {
           if (table_index[temp->c_id-1]==parent->c_id)
               { parent_choosen=parent;
                 ppid=parent->c_id;
               }
           else if (table_index[parent->c_id-1]==temp->c_id)
              { parent_choosen=temp;
                ppid=temp->c_id;
              }
               else { mergetag=0;
                      break;
                     }
         }
   }
   else if  ((ppid!=-99)&&(temp!=parent)&&
             (table_index[temp->c_id-1]!=ppid)) {
             mergetag=0;
             break;
           }
  temp=temp->next_sibling;
}

if (mergetag&&(parent_choosen!=NULL)) parent=parent_choosen;
tp3=pre=NULL;
par_id=-99;
tbok=1;
temp= parent->parent->left_child;
while (temp!=NULL) {
 pre=temp;
 if (temp->c_id<=0) {
   mergetag=0;
   tbok=0;
   break;
   }
 if  (par_id==-99) par_id =table_index[temp->c_id-1];
   else   if (table_index[temp->c_id-1]!=par_id) tbok=0 ;

 if ((temp->left_child==NULL)&&
      (table_index[temp->c_id-1]!=parent->c_id)) {
      mergetag=0;
      tbok=0;
      break;
```

```
    }

   temp=temp->next_sibling;
   if(temp==parent) tp3=pre;
   }

  if ((tbok)&&(par_id!=-99)) {
    tp2->c_id=par_id;
    mergetag=0;
  }

 if(mergetag) {
    /*change parent info */
    if (tp3==NULL){
       temp=parent->left_child;
       parent->parent->left_child=temp;

       while (temp!=NULL) {
           pre=temp;
           temp->parent=parent->parent;
           temp=temp->next_sibling;
           }
           pre->next_sibling=parent->next_sibling;
           parent->parent->c_id=parent->c_id;
           free(parent);
        }
    else {
       tp3->next_sibling=parent->left_child;
       while (tp3!=NULL) {
       temp=tp3;
       tp3->parent=parent->parent;
       tp3=tp3->next_sibling;
    }
   if (temp!=NULL) temp->next_sibling= parent->next_sibling;
        parent->parent->c_id=parent->c_id;
        free(parent);
   }
 } /*merge */
}
else {
    mergetag=0;
    pre=NULL;
    temp= parent->parent->left_child;
    while (temp!=NULL) {
        if(temp==parent) {
```

```
                tp3=pre;
                break;
              }
            temp=temp->next_sibling;
            pre=temp;
        }

        if(parent->c_id==parent->parent->c_id) mergetag=1;
        if(mergetag) {
             /*change parent info */
             if (tp3==NULL){
                 temp=parent->left_child;
                 parent->parent->left_child=temp;

                 while (temp!=NULL) {
                    pre=temp;
                    temp->parent=parent->parent;
                    temp=temp->next_sibling;
                 }
                 pre->next_sibling=parent->next_sibling;
                 parent->parent->c_id=parent->c_id;
                 free(parent);
              }
              else {
                  tp3->next_sibling=parent->left_child;
                  while (tp3!=NULL) {
                     temp=tp3;
                     tp3->parent=parent->parent;
                     tp3=tp3->next_sibling;
                   }
                 if (temp!=NULL)
                      temp->next_sibling= parent->next_sibling;
                      parent->parent->c_id=parent->c_id;
                      free(parent);
              }
         }
   }

return(tp2);
}

struct Tree * merge_neighbor(struct Tree *T)
{
   struct Tree *temp,*parent,*pre,*tp2;
   int i=0;
```

```
int id=-99, final=0;

if (T!=NULL){
  if(T->left_child==NULL)
  { if ((T ->parent!=NULL)&&(T ->parent->c_id<0)) {
      temp=T->parent->left_child;
      final=0;
      if ((id==-99)) id=table_index[temp->c_id-1];
         while (temp!=NULL) {
           if (id!=table_index[temp->c_id-1]) final=1;
           temp=temp->next_sibling;

          }
          if ((final!=1)&&(id!=-99))
            if (T->parent->c_id<0)  T->parent->c_id=id;
             to_Harvard(T->parent);
             to_Harvard(T2);
   }
  return T;
  }
  if (T->c_id<0) {
     temp=T->left_child;
     if ((id==-99)) id=table_index[temp->c_id-1];
     to_Harvard(T2);
     while (temp!=NULL) {
          if (id!=table_index[temp->c_id-1]) final=1;
          tp2=temp->next_sibling;
          merge_neighbor(temp);
          temp=tp2;
     }
     if(T->c_id>0)
         if (T->parent!=NULL) return(mark_one(merge(T)));

     if ((final!=1)&&(id!=-99))
       { if (T->c_id<0) T->c_id=id;
        /*  merge node with the same cid*/
           if (T->parent!=NULL) return(mark_one(merge(T)));
       }
      }
     else /*t->c_id>0 */{
        temp=T->left_child;
         while (temp!=NULL) {
          tp2=temp->next_sibling;
          merge_neighbor(temp);
          temp=tp2;
```

```
        }
       if (T->parent!=NULL) return(mark_one(merge(T)));
      }
    return(T);
  }
  return NULL;
}

struct Tree * mark_one(struct Tree *T)
{
  struct Tree *temp,*parent,*pre,*tp2;
  int i=0;
  int id=-99, final=0;

  if ((T!=NULL)&&(T ->c_id<0)&&(T->left_child!=NULL)){
        temp=T->left_child;
         final=0;
         if ((id==-99)) id=table_index[temp->c_id-1];
           while (temp!=NULL) {
             if (id!=table_index[temp->c_id-1]) final=1;
             temp=temp->next_sibling;
             }
         if ((final!=1)&&(id!=-99))
            T->c_id=id;
              to_Harvard(T->parent);
              to_Harvard(T2);
     }
    return T;
}


int pop_shink(struct Tree *T)
{
 struct Tree *temp;

 temp=T->parent;
 while (temp!=NULL) {
        if (temp->tag_marked==1) break;
        temp=temp->parent;
        }
 if((temp!=NULL)&&(temp->numchild==1))
    T->vertex_no=temp->vertex_no;/*T->tag_marked=0;*/
}

void shrink(struct Tree *T)
```

```
{
  struct Tree *temp;

  if (T!=NULL) {
   if ( (temp=T->left_child)==NULL) return ;
   else{
      while (temp!=NULL) {
      shrink(temp);
      temp=temp->next_sibling;
      }
   if ((strlen(T->info)==0)&&(T->tag_marked==1)) pop_shink(T);
    }
  }
}


char get_next_char()
{ if(!feof(fp_h)) return(fgetc(fp_h));
}

/*********************************************************************
/* 1 '('
/* 2 ')'
/* 3 string
/* 4 :parent or )parent
/* 5 end_of_line
/* 0 end_of_file
/*********************************************************************/

int get_next_token(char * str,int mode)
{
  static char ch_temp;
  char temp[256];
  char ch;
  static int i=0;
  static int temp_tag=0;
  static int parent_tag=0;

  if (mode==0) {
   i=0;
   temp_tag=0;
   parent_tag=0;

   if (ch_temp=='(') {
      temp_tag=1 ;
```

```
    return 1;
  }
  return 8;
 }

 if (temp_tag==0)
 {
  if(feof(fp_h)) return 0;
  ch=' ';
  while ((ch == ' ')||(ch == '\t')||(ch==',')) {
     if (ch==',') parent_tag=0;
     ch=get_next_char();
  }
  if(ch=='\n') return 5;

  if (ch=='(') {
     temp_tag=0 ;
     return 1;
  }
  else if (ch==')') {
      temp_tag=0;
      return 2;
  }
  else {
    while( !feof(fp_h)&&((ch!='(') &&(ch!='\n')&&
        (ch!='\t')&&(ch !=',')&& (ch!=' ')&&(ch!=')')) ){
      temp[i]=ch;
      ch=get_next_char();
      i++;
    }

  if ((ch=='(') ||(ch ==')')) temp_tag=1;
  ch_temp=ch;
  if (temp[i-1]=='\n') temp[i-1]='\0';
  temp[i]='\0';
  strcpy(str,temp);
  i=0;
  if (ch==',') temp_tag=0;
  if(parent_tag) {
     parent_tag=0;
     return 4;
     }
  else return 3;
 }
}
```

```
 else {
  if (ch_temp=='(') {  parent_tag=0;
                              temp_tag=0 ;
                              return 1;
                       }
       if (ch_temp==')') {
                              parent_tag=1;
                              temp_tag=0;
                              return 2;
                       }
       }
}



int check_order(char *fn)
{
 int rt_value;
 int first_time,l_count,r_count;

 if ((fp_h=fopen(fn,"r"))==NULL)
    { printf("\n Can not open file %s",fn);
      exit(0);
      }

  rt_value=get_next_token_ck(node_info);

  if (rt_value!=1) return (-1);

  depth=0;
  first_time=1;
  l_count=1;
  r_count=0;

  while (((rt_value=get_next_token_ck(node_info))!=0)) {
    if (first_time) {
         if (rt_value==1)
             l_count++;
         else first_time=0;
       }

    if (rt_value==1)    depth++;

    if (rt_value==2) {
        depth--;
```

```
        r_count++;
       }
    else r_count=0;

   }
 fclose(fp_h);

 if(depth!=-1) return (-1);
 if(l_count>=r_count) return (1);
   else return (2);
}

void construct_tree_preord(struct  Tree *T)
{
  int rt_value;
  struct Tree *current,*parent,*rmost_child,*temp;
  int current_level;

  rt_value=get_next_token(node_info,2);

  if (rt_value!=1) return ;

  while ((rt_value=get_next_token(node_info,2))==1) ;

  strcpy(T->info,node_info);

  T->parent=NULL;
  T->left_child=NULL;
  T->next_sibling=NULL;
  T->vertex_no=vertexno;
  vertexno++;

  depth=0;
  current_level=depth;
  current=T;
  parent=NULL;

  while (((rt_value=get_next_token(node_info,2))!=0)
        ||(depth != -1))
  {
    if (rt_value==1) {
       depth++;
       parent=current;
       }
```

```
    if (rt_value==2) {
       depth--;
       current=parent;
       if(depth!=-1)  parent=parent->parent;
     }
    if (rt_value==3) {
      if ((temp=(struct Tree *)mallocT())==NULL)
        { printf("Allocation memory error \n");
         exit(0);
        }
        strcpy(temp->info,node_info);
        current=temp;
        temp->parent=parent;
        temp->left_child=NULL;
        temp->next_sibling=NULL;
        temp->vertex_no=vertexno;
        vertexno++;

        if ((rmost_child=parent->left_child) == NULL)
           parent->left_child =temp;
        else {  while (rmost_child->next_sibling !=NULL)
                    rmost_child=rmost_child->next_sibling;
                rmost_child->next_sibling=temp;
             }
       }
    }
}


void construct_tree_pstord(struct  Tree *T)
{
  int rt_value;
  struct Tree *current,*parent,*rmost_child,*temp;
  int current_level;
  int tag=0;
  int finished;
  int first_time=1;

  T->parent=NULL;
  T->left_child=NULL;
  T->next_sibling=NULL;
  strcpy(T->info,"");
  T->vertex_no=vertexno;
  vertexno++;
```

```
depth=0;
current_level=depth;
current=T;
parent=T;
finished=0;

rt_value=get_next_token(node_info,2);
while (( rt_value !=0)&&(!finished) ) {
  if(rt_value==5) return;
  if (rt_value==1) {
    tag=0;
    if ((temp=(struct Tree *)mallocT())==NULL)
      { printf("Allocation memory error \n");
        exit(0);
        }

   temp->parent=parent;
   temp->left_child=NULL;
   temp->next_sibling=NULL;
   temp->vertex_no=vertexno;
   strcpy(temp->info,"");

   if ((rmost_child=parent->left_child) == NULL)
       parent->left_child =temp;
    else {
       while (rmost_child->next_sibling !=NULL)
          rmost_child=rmost_child->next_sibling;
       rmost_child->next_sibling=temp;
    }
    vertexno++;
    depth++;
    parent=temp;
 }

 if (rt_value==2) {
    tag=0;
    depth--;
    if(depth==0)  finished=1;

    current= parent;
    parent=parent->parent;
  }

 if (rt_value==4) strcpy(current->info,node_info);
```

```
   if (rt_value==3) {
      tag=1;
      if (tag==1) {
        if ((temp=(struct Tree *)mallocT())==NULL)
         { printf("Allocation memory error \n");
          exit(0);
          }

       strcpy(temp->info,node_info);
       temp->parent=parent;
       temp->left_child=NULL;
       temp->next_sibling=NULL;
       temp->vertex_no=vertexno;

       if ((rmost_child=parent->left_child) == NULL)
          parent->left_child =temp;
        else {  while (rmost_child->next_sibling !=NULL)
                   rmost_child=rmost_child->next_sibling;
                rmost_child->next_sibling=temp;
          }
        vertexno++;
      }
      else strcpy(current->info,node_info);
     tag=1;
    }
    rt_value=get_next_token(node_info,2);
  }
  if ( rt_value==4) strcpy(current->info,node_info);
}

void construct_tree_prord(struct  Tree *T)
{
  int rt_value;
  int current_level;
  int tag=0;
  struct Tree *current,*parent,*rmost_child,*temp;

  rt_value=get_next_token(node_info,2);

  if (rt_value!=1) return ;

  T->parent=NULL;
  T->left_child=NULL;
  T->next_sibling=NULL;
  strcpy(T->info,"");
```

```
T->vertex_no=vertexno;
vertexno++;

depth=0;
current_level=depth;
current=T;
parent=NULL;

while (((rt_value=get_next_token(node_info,2))!=0)
      ||(depth != -1))
  {
  if (rt_value==1) {
      tag=0;
      if ((temp=(struct Tree *)mallocT())==NULL)
       { printf("Allocation memory error \n");
        exit(0);
        }

      temp->parent=current;
      temp->left_child=NULL;
      temp->next_sibling=NULL;
      temp->vertex_no=vertexno;
      if ((rmost_child=current->left_child)
         == NULL) current->left_child =temp;
       else {
         while (rmost_child->next_sibling !=NULL)
           rmost_child=rmost_child->next_sibling;
         rmost_child->next_sibling=temp;
         }
     vertexno++;
     depth++;
     current=temp;
     parent=current;
     }

  if (rt_value==2) {
     tag=0;
     depth--;
     current=current->parent;
   if(depth!=-1)  parent=parent->parent;
  }
  if (rt_value==3) {
    if (tag==1) {
      if ((temp=(struct Tree *)mallocT())==NULL)
       { printf("Allocation memory error \n");
```

```
          exit(0);
          }
        strcpy(temp->info,node_info);

        temp->parent=current->parent;
        temp->left_child=NULL;
        temp->next_sibling=NULL;
        temp->vertex_no=vertexno;

        rmost_child=current;
        while (rmost_child->next_sibling !=NULL)
          rmost_child=rmost_child->next_sibling;
        rmost_child->next_sibling=temp;

       vertexno++;
       current=temp;
       parent=current->parent;
       }
       else strcpy(current->info,node_info);
      tag=1;
      }
   }
}

int to_pd_tree(struct Tree *T)
{
 struct Tree *temp;

 if ((temp=T->left_child)!=NULL) {
     printf( "%10s | ",treeid);
     printf( " %4d |",T->vertex_no);
     while (temp!=NULL) {
        printf( " %4d ",temp->vertex_no);
        temp=temp->next_sibling;
        }
     temp=T->left_child;
     while (temp!=NULL) {
        to_pd_tree(temp);
        temp=temp->next_sibling;
        }
    }
 }


void mark_labelq(struct Tree *T)
```

```
{

 struct Tree *temp;

  if (T!=NULL) {
       T->tag_marked=tableq[T->vertex_no-1][1];
   T->vertex_no=re_count;
   re_count++;
     }

 if ((temp=T->left_child)!=NULL) {
       while (temp!=NULL) {
        mark_labelq(temp);
        temp=temp->next_sibling;
        }
    }
}

int to_pd_label(struct Tree *T,char ndl[MAXNODE][256])
{
 struct Tree *temp;

  if (T!=NULL) {
      strcpy(ndl[T->vertex_no-1],T->info);
      if (T->parent !=NULL)
       table_index[T->vertex_no-1]=  (T->parent)->vertex_no;
      else table_index[T->vertex_no-1]= -1; /*root*/
      printf(" %10s | ",treeid);
      printf(" %4d |",T->vertex_no);
      printf("  %s  \n",T->info);
     }

 if ((temp=T->left_child)!=NULL) {
       while (temp!=NULL) {
        to_pd_label(temp,ndl);
        temp=temp->next_sibling;
        }
    }
 }

void  to_Harvard(struct Tree *T)
{
 struct Tree *temp;

 if (T!=NULL) {
```

```
  if ((temp=T->left_child)!=NULL) {
        printf("(");
        while (temp!=NULL) {
         to_Harvard(temp);
         temp=temp->next_sibling;
         if(temp!=NULL) printf(",");
         }
        printf(")%s%i", T->info,T->c_id);
     }
     else printf("%s%i", T->info,T->c_id);
  }
}

void  relabel(struct Tree *T)
{
 struct Tree *temp;
 int i;

  if (T!=NULL) {
      T->vertex_no=relabel_count;
      relabel_count++;
      if(T->c_id>0) {
         for (i=0;i< meaningful ;i++)
            if (table[i][0]==(T->c_id-1)) {
              table[i][1]=T->vertex_no-1;
              break;
             }
       }
     if ((temp=T->left_child)!=NULL) {
         while (temp!=NULL) {
         relabel(temp);
         temp=temp->next_sibling;
         }
     }
  }
}
```

REFERENCES

[1] E. N. Adams. Consensus techniques and the comparison of taxonomic trees. *Systematic Zoology*, 21:390–397, 1972.

[2] V. Berry and D. Bryant. Faster reliable phylogenetic analysis. In *Proceedings of the 3rd Annual International Conference on Computational Molecular Biology*, pp. 59–68, 1999.

[3] P. Buneman, S. B. Davidson, M. F. Fernandez, and D. Suciu. Adding structure to unstructured data. In *Proceedings of the 6th International Conference on Database Theory*, pp. 336–350, 1997.

[4] E. K. Brown and W. H. E. Day. A computationally efficient approximation to the nearest neighbor interchange metric. *Journal of Classification*, 1:93–124, 1984.

[5] G. S. Brodal, R. Fagerberg, and C. N. S. Pedersen. Computing the quartet distance between evolutionary trees in time $O(nlog^2n)$. In *Proceedings of the 12th Annual International Symposium on Algorithms and Computation*, pp. 731–742. Lecture Notes of Computer Science, Vol. 2223, Springer Verlag, Berlin, 2001.

[6] D. Bryant, J. Tsang, P. Kearney, and M. Li. Computing the quartet distance between evolutionary trees. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, San Francisco, CA, 2000.

[7] P. Buneman. The recovery of trees from measures of dissimilarity. In *Mathematics in Archaeological and Historical Sciences*, pp. 387–395. Edinburgh University Press, 1971.

[8] J. H. Camin and R. R. Sokal. A method for deducing branching sequences in phylogeny. *Evolution*, 19:311–326, 1965.

[9] J. H. Camin and R. R. Sokal. A method for deducing branching sequences in phylogeny. *Evolution*, 19:311–326, 1965.

[10] G. Chang, M. J. Healey, J. A. M. McHugh, and J. T. L. Wang. *Mining the World Wide Web: An Information Search Approach.* Kluwer Academic Publishers, Norwell, Massachusetts, 2001.

[11] S. S. Chawathe, S. Abiteboul, and J. Widom. Representing and querying changes in semistructured data. In *Proceedings of the IEEE International Conference on Data Engineering*, pp. 4–13, 1998.

[12] S. S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 26–37, 1997.

[13] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 493–504, 1996.

[14] Z. Chen, H. V. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. T. Ng, and D. Srivastava. Counting twig matches in a tree. In *Proceedings of the IEEE International Conference on Data Engineering*, pp. 595–604, 2001.

[15] R. Cole, M. Farach-Colton, R. Hariharan, T. M. Przytycka, and M. Thorup. An O(n log n) algorithm for the maximum agreement subtree problem for binary trees. *SIAM J. Comput.*, 30(5):1385–1404, 2000.

[16] R. Cole and R. Hariharan. An O(n log n) algorithm for the maximum agreement subtree problem for binary trees. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 323–332, 1996.

[17] B. DasGupta, X. He, T. Jiang, M. Li, J. Tromp, and L. Zhang. On distances between phylogenetic trees. In *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms*, pp. 427–436, 1997.

[18] B. DasGupta, X. He, T. Jiang, M. Li, J. Tromp, L. Wang, and L. Zhang. Computing distances between evolutionary trees. In D. Z. Du and P. M. Pardalos (eds.) *Handbook of Combinatorial Optimization*, Kluwer Academic Publishers, Volume 2, 1998, pp. 35–76.

[19] W. H. E. Day. Optimal algorithms for comparing trees with labeled leaves. *Journal of Classification*, 2:7–28, 1985.

[20] C. R. Douchette. An efficient algorithm to compute quartet dissimilarity measures. Unpubl. BSc (Hons) Dissertation, Memorial Univ. Newfoundland, 1985.

[21] A. Dress and M. Kruger. Parsimonious phylogenetic trees in metric spaces and simulated annealing. *Advances in Applied Mathematics*, 8:8–37, 1987.

[22] M. P. Evett, J. A. Hendler, A. Mahanti, D. S. Nau. PRA*: Massively Parallel Heuristic Search. *Journal of Parallel and Distributed Computing*, 25(2): 133-143, (1995.)

[23] M. Farach and M. Thorup. Fast comparison of evolutionary trees. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1994.

[24] M. Farach and M. Thorup. Optimal evolutionary tree comparison by sparse dynamic programming (extended abstract). In *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science*, pp. 770–779, 1994.

[25] J. Felsenstein. Evolutionary trees from DNA sequences: A maximum likelihood approach. *Journal of Molecular Evolution*, 17:368–376, 1981.

[26] A. Ferro, G. Gallo, R. Giugno, and A. Pulvirenti. Best-match retrieval for structured images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(7):707–718, 2001.

[27] W. Fitch. Toward the defining the course of evolution: Minimum change for a specific tree topology. *Systematic Zoology*, 20:406–416, 1971.

[28] O. Gotoh. An Improved Algorithm for Matching Biological Sequences. *J. Mol. Biol.*, 162, pp. 705-708, 1982.

[29] J. Hein, T. Jiang, L. Wang, and K. Zhang. On the complexity of comparing evolutionary trees. *Discrete Applied Mathematics*, 71:153–169, 1996.

[30] J. Jaakkola and P. Kilpelainen. Using sgrep for querying structured text files. University of Helsinki, Department of Computer Science, Report C-1996-83, November 1996; available at `http://www.cs.helsinki.fi/u/jjaakkol/sgrep.html`.

[31] H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A tree algebra for XML. In *Proceedings of the 8th Workshop on Data Bases and Programming Languages*, 2001.

[32] D. E. Joyce. Phylogeny and reconstructing phylogenetic trees. http://aleph0.clarku.edu/~djoyce/java/, 2000.

[33] S. Kannan, E. Lawler, and T. Warnow. Determining the evolutionary tree. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 475–484, 1990.

[34] S. Kannan, T. Warnow, and S. Yooseph. Computing the local consensus of trees. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1995.

[35] M. Kao, T. Lam, T. M. Przytycka, W. Sung, and H. Ting. General techniques for comparing unrooted evolutionary trees. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pp. 54–65, 1997.

[36] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48:96-129, 1998.

[37] E. Kubicka, G. Kubicki, and F. R. McMorris. An algorithm to find agreement subtrees. *Journal of Classification*, 12(1):91–99, 1995.

[38] T. W. Lam, W. K. Sung, and H. F. Ting. Computing the unrooted maximum agreement subtree in subquadratic time. In *Proceedings of the 5th Scandinavian Workshop on Algorithm Theory*, pp. 124–135, 1996.

[39] T. W. Leung, G. Mitchell, B. Subramanian, B. Vance, S. L. Vandenberg, and S. B. Zdonik. The AQUA data model and algebra. In *Proceedings of the 4th Workshop on Data Bases and Programming Languages*, pp. 157–175, 1993.

[40] T.-L. Liu and D. Geiger. Approximate Tree Matching and Shape Similarity. In *Proceedings of IEEE International Conference on Computer Vision* vol. 1, pp. 456-462, 1999.

[41] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 319–327, 1990.

[42] G. Nelson. Cladistic analysis and synthesis: Principles and definitions, with a historical note on Adanson's Famille Des Plantes (1763-1764). *Syst. Zoology*, 28:1–21, 1979.

[43] A. S. Noetzel and S. M. Selkow. An analysis of the general tree-editing problem. In D. Sankoff and J. B. Kruskal, editors, *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, pp. 237–252. Addison-Wesley, Reading, MA, 1983.

[44] NSF Workshop Report at Yale University. Assembling the tree of life: Research needs in phylogenetics and phyloinformatics, July 2000.

[45] R. D. M. Page. Comments on component-compatibility in historical biogeography. *Cladistics*, 5:167–182, 1989.

[46] R. D. M. Page. COMPONENT. http://taxonomy.zoology.gla.ac.uk/rod/ cpw.html, 2003.

[47] R. D. M. Page and M. A. Charleston. Trees within trees: Phylogeny and historical associations. *Trends in Ecology and Evolution*, 13:356–359, 1998.

[48] R. D. M. Page and E. C. Holmes. *Molecular Evolution: A Phylogenetic Approach*, Blackwell Science,1998.

[49] M. Pelillo. Matching free trees, maximal cliques, and monotone game dynamics. In *the IEEE Transactions on Pattern Analysis and Machine Intelligence.* 24(11):1535-1541, 2002.

[50] W. H. Piel, M. J. Donoghue, and M. J. Sanderson. TreeBASE: A database of phylogenetic information. In *Proceedings of the 2nd International Workshop of Species 2000*, 2000.

[51] M. Rarey and J. S. Dixon. Feature Trees: A new molecular similarity measure based on tree matching. *Journal of Computer-Aided Molecular Design* 12, pp. 471-490, 1998.

[52] M. J. Sanderson, M. J. Donoghue, W. H. Piel, and T. Eriksson. TreeBASE: A prototype database of phylogenetic analyses and an interactive tool for browsing the phylogeny of life. *American Journal of Botany*, 81(6):183, 1994.

[53] H. Shan, K. G. Herbert, W. H. Piel, D. Shasha, and J. T. L. Wang. A structure-based search engine for phylogenetic databases. In *Proceedings of the 14th International Conference on Scientific and Statistical Database Management*, pp. 7–10, 2002.

[54] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pp. 39–52, 2002.

[55] D. Shasha, J. T. L. Wang, H. Shan, and K. Zhang. ATreeGrep: Approximate searching in unordered trees. In *Proceedings of the 14th International Conference on Scientific and Statistical Database Management*, pp. 89–98, 2002.

[56] B. Subramanian, T. W. Leung, S. L. Vandenberg, and S. B. Zdonik. The AQUA approach to querying lists and trees in object-oriented databases. In *Proceedings of the IEEE International Conference on Data Engineering*, pp. 80–89, 1995.

[57] B. Subramanian, S. B. Zdonik, T. W. Leung, and S. L. Vandenberg. Ordered types in the AQUA data model. In *Proceedings of the 4th Workshop on Data Bases and Programming Languages*, pp. 115–135, 1993.

[58] TreeGen: Tree generation from distance data. Computational Biochemistry Research Group, ETH Zurich. Retrieved from `http://cbrg.inf.ethz.ch/Server/subsection3_1_6.html`, May, 2004.

[59] J. T. L. Wang, B. A. Shapiro, D. Shasha, K. Zhang, and K. M. Currey. An algorithm for finding the largest approximately common substructures of two trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):889–895, 1998.

[60] J. T. L. Wang, K. Zhang, G. Chang, and D. Shasha. Finding approximate patterns in undirected acyclic graphs. *Pattern Recognition*, 35(2):473–483, 2002.

[61] J. T. L. Wang, G.-W. Chirn, T. G. Marr, B. A. Shapiro, D. Shasha, and K. Zhang. Combinatorial pattern discovery for scientific data: Some preliminary results. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 115–125, 1994.

[62] J. T. L. Wang, K. Zhang, K. Jeong, and D. Shasha. A system for approximate tree matching. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):559–571, 1994.

[63] J. T. L. Wang, B. A. Shapiro, and D. Shasha (eds). *Pattern Discovery in Biomolecular Data: Tools, Techniques and Applications.* Oxford University Press, New York, 1999.

[64] L. Wang and D. Gusfield. Constructing Additive Trees When the Error Is Small. *Journal of Computational Biology* 5(1): 137–134, 1998.

[65] L. Wang, K. Zhang, and L. Zhang. Perfect phylogenetic networks with recombination. In *Proceedings of the ACM Symposium on Applied Computing*, pp. 46–50, 2001.

[66] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.

[67] K. Zhang, D. Shasha, and J. T. L. Wang. Approximate tree matching in the presence of variable length don't cares. *Journal of Algorithms*, 16(1):33–66, 1994.

[68] K. Zhang, R. Statman, and D. Shasha. On the editing distance between unordered labeled trees. *Information Processing Letters*, 42:133–139, 1992.

[69] K. Zhang, J. T. L. Wang, and D. Shasha. On the editing distance between undirected acyclic graphs. *International Journal of Foundations of Computer Science*, 7(1):43–57, 1996.